Títol:	OpenMP to OpenCL: Aprovechamiento de los recursos heterogéneos del sistema
Autor:	Guillén Allés, Moisés
Data:	30 de Junio
Director:	Jiménez González, Daniel
Departament del director:	Departament d'Arquitectura de Computadors
Titulació:	Enginyeria Informàtica
Centre:	Facultat d'Informàtica de Barcelona (FIB)
Universitat:	Universitat Politècnica de Catalunya (UPC) BarcelonaTech

DADES DEL PROJECTE

Títol del Projecte:	OpenMP to OpenCL: Aprovechamiento de los recursos heterogéneos del sistema
Nom de l'estudiant:	Guillén Allés, Moisés
Titulació:	Enginyeria en Informàtica
Crèdits:	37.5
Director/Ponent:	Jiménez González, Daniel
Departament:	Departament d'Arquitectura de Computadors

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President:	Álvarez Martínez, Carlos
Vocal:	Brunat Blay, Josep Maria
Secretari:	Jiménez González, Daniel

QUALIFICACIÓ

Qualificació numèrica:
Qualificació descriptiva:

Data:

Facultat d'Informàtica de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Universitat Politècnica de Catalunya - Facultat d'Informàtica de Barcelona
Enginyeria Informàtica

OpenMP to OpenCL: Aprovechamiento de los recursos heterogéneos del sistema

Departament d'Arquitectura de Computadors

Director: Jiménez González, Daniel
Alumno: Guillén Allés, Moisés

Barcelona, 2011

Agradecimientos

A German Llort, del grupo de *Performance Tools* del *BSC*, por resolvernos las dudas para el funcionaban las aplicaciones de instrumentación.

A Roger Ferrer, del grupo de *Programming Models* del *BSC*, por ayudarnos a usar el *Mercurium*

A Xavier Martorell, Javier Teruel y Alex Duran del grupo de desarrolladores del proyecto *Nanos* por la ayuda recibida para entender el funcionamiento de *Nanos*.

A Daniel Jimenez, por toda su ayuda y su paciencia ya que este proyecto no habría sido posible sin él.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

Glosario

A

Application programming interface (API) Interfaz de programación de aplicaciones, pág. 13.

C

C Lenguaje de programación popular de propósito general, pág. 5.

C++ Lenguaje de programación popular basado en *C* con soporte para la programación orientada a objetos, pág. 15.

Cell Broadband. Engine (Cell BE) microprocesador diseñado para cubrir el hueco existente entre procesadores convencionales y los procesadores especializados de alto rendimiento como las *GPU*s. La arquitectura fue desarrollada conjuntamente por *Sony Computer Entertainment*, *Toshiba* e *IBM.*, pág. 43.

Central Processing Unit (CPU) Unidad central de procesamiento conocido como procesador o multiprocesador, pág. 6.

compilador Programa informático que traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación, pág. 16.

Compute Unified Device Architecture (CUDA) compilador y un conjunto de herramientas creadas por *nVidia* que permiten a los programadores ejecutar código en *GPU*s de *nVidia*, pág. 25.

F

framework Estructura conceptual y tecnológica de soporte definida, normalmente con artefactos o módulos de software concretos, con base en la cual otro proyecto de software puede ser organizado y desarrollado, pág. 12.

G

Graphics Processing Unit (GPU) Procesador dedicado al procesamiento de gráficos, pág. 6.

O

Open Computing Language (OpenCL) *framework* para escribir y ejecutar programas en diversas plataformas, como *GPU*s,*CPU*s y otros procesadores, pág. 5.

Open Multi-Processing (OpenMP) Modelo de programación paralela de memoria compartida, pág. 5.

OpenMP + StarSs (OmpSs) Extensión de *OpenMP* para aplicar el paradigma en sistemas heterogeneos. El origen del nombre proviene de la fusión de *OpenMP* y *StarSs*, pág. 5.

P

pragma Directivas para el compilador, para añadir algún tipo de información, pág. 15.

R

runtime Entorno de ejecución, pág. 16.

S

script Son archivos de ordenes para facilitar tareas que se suelen hacer en un terminal., pág. 47.

Software Development Kit (SDK) Conjunto de herramientas de desarrollo que le permite a un programador crear aplicaciones para un sistema concreto., pág. 43.

StarSs (StarSs) Modelo de programación basado en tareas, aprovechando un sistema heterogeneo, pág. 12.

T

thread-safe Concepto de programación que indica que un programa puede ser ejecutado simultáneamente con varios hilos de ejecución, pág. 59.

Capítulo 1

Introducción

La tendencia actual en el diseño de microprocesadores es reducir el consumo de energía, siendo una de las vías evitar el aumento de la frecuencia a la que trabajan los procesadores. Para lograr este objetivo, sin sacrificar potencia de cálculo, algunos procesadores incorporan varios núcleos (*CPU*s). De esta forma, una aplicación podría llegar a explotar el paralelismo a nivel de tareas reduciendo el tiempo total de ejecución. Aún así, el uso eficiente de estos núcleos, y por consiguiente, de consumo de energía, conlleva un esfuerzo grande por parte del programador.

Otra vía de aprovechar mejor el consumo de energía de un sistema es el uso de las *GPU*s para hacer cómputos de carácter general. Éstas tienen una capacidad alta de cálculo y se podrían usar cuando no estuvieran ocupadas; aprovechando mejor los recursos del sistema. De hecho, en los últimos años, el uso de las *GPU*s para cómputos de carácter general se ha visto beneficiado por la aparición de *API*s, proporcionadas por los fabricantes de los dispositivos, que facilitan su uso.

Ambas posibilidades: más de un núcleo en el procesador y uso de las *GPU*s, nos llevan a la necesidad de aprovechar mejor todos esos recursos en la ejecución de nuestras aplicaciones. Para alcanzar este objetivo necesitamos que la aplicación tenga secciones que se puedan ejecutar como tareas, posiblemente en paralelo[14]. Además,

el programador y/o el compilador deberían ser capaces de expresar y explotar este paralelismo para los recursos del los que dispone, lo cual puede ser sumamente complejo.

A todo esto se le suma el hecho que para utilizar cualquier tipo de dispositivo, como una *GPU*, el programador tiene que aprender cómo acceder a ella. El mantenimiento de una aplicación de este tipo conlleva un coste elevado y una complejidad que puede ser inviable para muchos proyectos.

1.1. Objetivos del proyecto

El proyecto pretende facilitar la tarea del programador para que su aplicación aproveche todos los recursos del sistema, dándoles un paradigma de programación en paralelo con capacidad de ejecutar en *GPU*s y otros dispositivos.

1.1.1. Objetivo principal

El objetivo principal de nuestro proyecto es diseñar y implementar un modelo de programación basado en *OpenMP 3.0* extendido (*OmpSs*) sobre *OpenCL*.

OpenMP es un modelo de programación paralela de memoria compartida y basado en directivas[5]. Este modelo de programación añade simplicidad y flexibilidad para ejecutar procesos en diversas plataformas. Básicamente, *OpenMP* consiste en extender un lenguaje de programación, en nuestro caso *C* y *C++*, mediante *pragma*s, especificando partes del código donde el programa puede ejecutarse en paralelo, puntos de sincronización, políticas de distribución de trabajo, etc.

Figura 1.1: Logo de *OpenMP*

OmpSs es una extensión de *OpenMP* que permite al programador expresar las dependencias de datos entre tareas (*tasks* en *OpenMP* 3.0). Con ello se consigue, en tiempo de ejecución y de forma transparente al programador, que aquellas tareas que no tengan sus datos preparados se esperen a tenerlos, y una vez los tienen, se ejecuten. *OmpSs* también permite especificar un conjunto de dispositivos donde se puede ejecutar una o varias partes del programa[9].

OpenCL es un estándar abierto para la programación de aplicaciones paralelas sobre unidades de computación heterogéneas, como *GPU*s, *CPU*s y otros dispositivos de

aceleración[1]. Su estructura consta de dos partes: *a*) Una *API* para acceder y usar las unidades de computación, y *b*) Un lenguaje de programación que se ejecutará dentro del dispositivo o unidad de proceso.

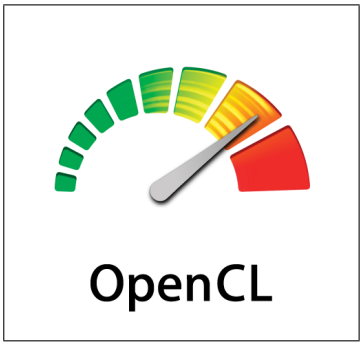

Figura 1.2: Logo de *OpenCL*

Para conseguir los objetivos de nuestro proyecto, éste se divide en dos partes: *a*) Un *compilador source-to-source*[2] y *b*) el *Runtime*.

En particular, el programador utilizará *pragma*s de *OmpSs* en su aplicación para expresar paralelismo y en qué dispositivos se puede ejecutar una determinada tarea. El compilador interpretará los *pragma*s del usuario y generará código fuente que incluye llamadas a nuestra librería de soporte a la ejecución, el *runtime*. Nuestro *runtime* implementará las llamadas a *OpenCL* utilizando los recursos del los dispositivos disponibles. La figura 1.3 muestra una esquema del proyecto. El código transformado por el compilador *source-to-source*, y que contiene llamadas al *runtime*, se ejecutará en el procesador que recibe el nombre de procesador *host*. Además, se generará un fichero de código fuente por cada parte de código a acelerar. Cada una de estas partes a acelerar recibe el nombre de *kernel* y está escrita en lenguaje *C* de *OpenCL*. Estos *kernels* se tienen que compilar para un determinado dispositivo en tiempo de ejecución. Para ello se usará la librería de *OpenCL*. Posteriormente, los *kernels* se ejecutarán en algunos de los dispositivos soportados por *OpenCL*: *GPU*s, *CPU*s y otros dispositivos.

[1] Por ejemplo los *IBM Cell Blade*.
[2] Código fuente a código fuente. Este tipo de compiladores no genera objetos ni binarios ejecutables

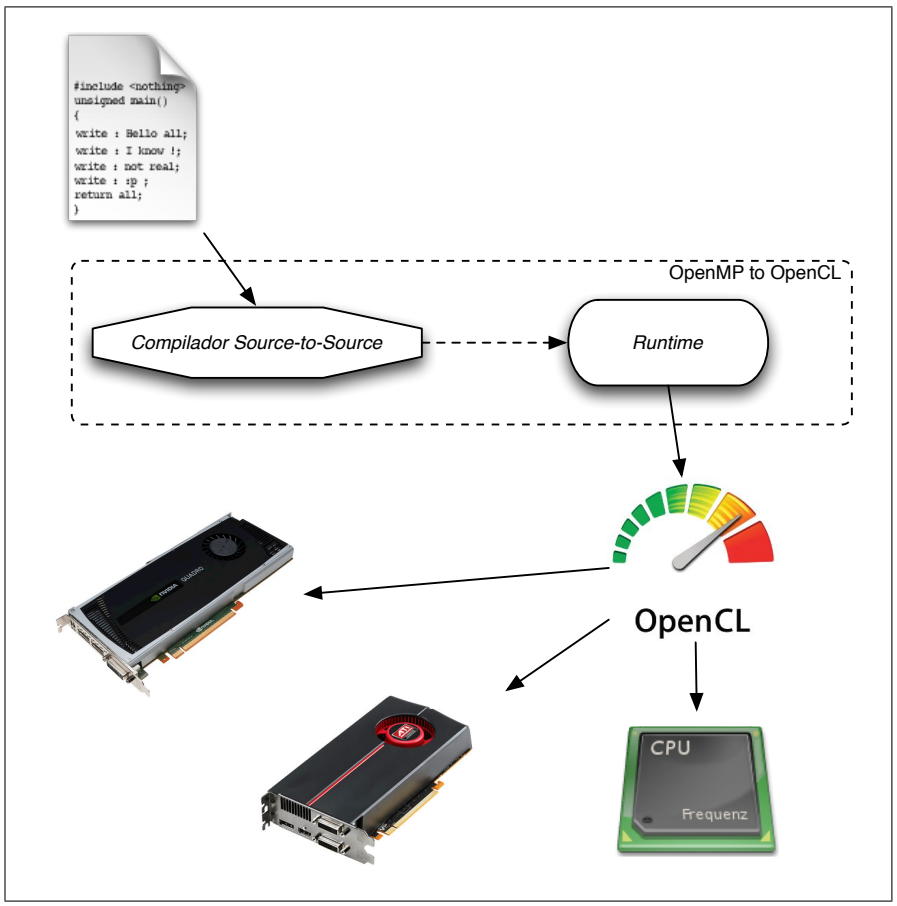

Figura 1.3: Esquema básico del proyecto

1.1.2. Subobjetivos

Este objetivo principal lo podemos dividir en una lista más detallada de subobjetivos que listamos a continuación:

1. Dar soporte a las cláusulas y directivas de *OpenMP* extendido (*OmpSs*):

   a) Cláusula *target device*: Interpretar y obtener la información de los dispositivos en los que el programador permite ejecutar ese proceso/tarea

   b) Cláusula *task*: Obtener información de los argumentos y dependencias a considerar en la ejecución de ese proceso/tarea.

   c) Cláusula *taskwait*: Marcar un punto de sincronización.

2. Transformar automáticamente los fragmentos de código a acelerar en códigos ejecutables dentro del entorno de *OpenCL*.

3. Generar automáticamente las llamadas necesarias para utilizar el *runtime*, que permita la creación de tareas a partir de las funciones a acelerar.

    a) Generar código *OpenCL* para las funciones a acelerar.

    b) Realizar la entrada y salida de los argumentos.

    c) Especificar el tipo de argumentos.

    d) Identificar y utilizar la información de la dependencia de datos.

    e) Crear los puntos de sincronización.

    f) Determinar y gestionar los dispositivos en los que se tiene que ejecutar la tarea.

4. Planificar adecuadamente la ejecución de las tareas:

    a) Determinar y gestionar las dependencias.

    b) Gestionar los procesos pendientes.

5. Controlar los recursos de *OpenCL*:

    a) Inicializar dispositivos.

    b) Gestionar las memorias disponibles.

    c) Crear objetos de memoria.

    d) Crear estructuras para la ejecución.

6. Verificar el funcionamiento y valorar el rendimiento.

7. Escribir la memoria del proyecto.

Nótese que los objetivos del 1 al 3 pertenecen al compilador *source-to-source*, y los objetivos 4 y 5 son parte del *runtime*.

1.2. Metodología

En primer lugar se han seleccionado las construcciones y cláusulas de *OmpSs* que permitieran alcanzar nuestros objetivos. Después hemos determinado qué funcionalidades y llamadas a la *API* del *OpenCL* eran necesarias en base a las construcciones seleccionadas previamente. Tras este paso, hemos realizado comprobaciones manuales de las principales funcionalidades que necesitábamos del estándar *OpenCL*. Una vez averiguamos la funcionalidad real que ofrecían las dos implementaciones de *OpenCL* existentes, implementamos manualmente la traducción de las construcciones y cláusulas a las funcionalidades que mejor se acercaban a nuestros objetivos. Finalmente, automatizamos el proceso introduciendo los pasos necesarios en el compilador *source-to-source*.

Una vez llegados a este punto, debíamos introducir el control de dependencias con tal de que las tareas se pudieran lanzar de forma adecuada en cuanto tuvieran sus entradas disponibles. Se realizó un análisis de las posibles implementaciones y la solución más adecuada, en tiempo de implementación y eficiencia de ejecución, era integrar nuestro compilador con el *runtime* de *Nanos* con tal de aprovechar su control de dependencias. La gestión y control de dispositivos la continuamos realizando con nuestro *runtime* por debajo de *Nanos*.

El test de funcionalidad de nuestro sistema se ha desarrollado en dos etapas, la primera de ellas durante el desarrollo del proyecto para determinar/verificar cada funcionalidad incorporada, y la segunda de ellas para analizar el rendimiento de las aplicaciones.

Finalmente se decidió añadir un sistema de generación de trazas con tal de analizar detalladamente la ejecución de las aplicaciones en los diferentes dispositivos.

En la presente memoria no detallaremos cada una de las fases del desarrollo para cada prototipo, sino que abordaremos directamente la versión final. Solo se harán

referencias a algunos resultados intermedios obtenidos que sean concluyentes en el diseño final.

Capítulo 2

OmpSs: OpenMP + StarSs

OmpSs es un modelo de programación basado en *OpenMP* 3.0[6] que incorpora las características más importantes de *StarSs*, desarrollado por el *BSC*[1][9].

StarSs añade a *OpenMP* la posibilidad de especificar dependencias entre las tareas mediante la definición de entradas y salidas que afectan a la tarea, y expresar explícitamente las transferencias de memoria necesarias para ejecutar dicha tarea. Además, el *runtime* de *OmpSs* permite gestionar automáticamente qué tareas pueden ejecutarse una vez sus entradas están disponibles y hay recursos para poder ejecutarse. Este *runtime* está hecho sobre *Nanos*.

A continuación especificamos las construcciones de *OmpSs* que nos afectan en este proyecto:

- Construcción **task**: Indica que se debe crear una tarea para ejecutar el código que está bajo su contexto. Esta construcción permite una serie de cláusulas para especificar dependencias:
  - **input**: Indica una dependencia de entrada de datos.
  - **output**: Indica una dependencia de salida de datos.

[1] Barcelona Supercomputing Center

    - **inout**: Indica una dependencia de entrada y salida de datos.

- Construcción **taskwait**: Evita continuar ejecutando el código hasta que no hayan terminado las tareas pendientes.

- Construcción **target**: Soporte para especificar los dispositivos y las transferencias de memoria necesarias. Las cláusulas disponibles son:

    - **device**: Conjuntos de dispositivos donde se puede ejecutar la tarea.
    - **copy_in**: Indica que los datos se tiene que transferir al dispositivo.
    - **copy_out**: Indica que una vez terminada la tarea, hay que transferir dichos datos a nuestra memoria.
    - **copy_inout**: Combinación de las dos cláusulas *copy_in* y *copy_out*.
    - **copy_deps**: Interpreta las cláusulas de dependencia como transferencias de memoria.
    - **implements**: Especifica que el código esta implementado para un dispositivos en concreto indicado con la cláusula *device*

El código 2.1 muestra un ejemplo con estas construcciones y cláusulas. En este ejemplo hemos definido tres funciones. Para cada una de ellas, y utilizando las cláusulas de la construcción *task*, hemos expresado las dependencias de memoria que le afectan (*inputs*, *inouts*) y las dependencias de memoria que afectan a otras tareas (*outputs*, *inouts*). Con la definición de estas dependencias y la cláusula *copy_deps* de la construcción *target* se está indicando, además, las transferencias necesarias entre el procesador *host* y los dispositivos.

Las llamadas a dichas funciones se convertirán en tareas que se ejecutarán dentro del dispositivo indicado.

```
1 #pragma omp target device(smp) copy_deps
2 #pragma omp task input(A, B) output(C)
3 void vadd3 (float A[BS], float B[BS],float C[BS]);
```

```
4
5  #pragma omp target device(cuda) copy_deps
6  #pragma omp task input(sum, A) inout(B)
7  void scale_add (float sum, float A[BS],float B[BS]);
8
9  #pragma omp target device(smp) copy_deps
10 #pragma omp task input(A) inout(sum)
11 void accum (float A[BS], float *sum);
12
13 ...
14 for (i=0; i<N; i+=BS)                    // C=A+B
15    vadd3 ( &A[i], &B[i], &C[i]);
16 ...
17 for (i=0; i<N; i+=BS)                    // sum(C[i])
18    accum (&C[i], &sum);
19 ...
20 for (i=0; i<N; i+=BS)                    // B=sum*A
21    scale_add (sum, &E[i], &B[i]);
22 ...
23 for (i=0; i<N; i+=BS)                    // A=C+D
24    vadd3 (&C[i], &D[i], &A[i]);
25 ...
26 for (i=0; i<N; i+=BS)                    // E=G+F
27    vadd3 (&G[i], &F[i], &E[i]);
28 ...
29 #pragma omp taskwait
30 ...
```

Código 2.1: Ejemplo OmpSs: declaración de los pragmas

El *runtime* del *OmpSs* garantiza que se respetará las dependencias en la ejecución de tareas, planificando cómo se deben ejecutar las tareas y cómo se sincronizarán con los dispositivos utilizados. En la figura 2.1 vemos una posible secuencia de ejecución de las tareas en el código 2.1, indicada por la numeración en la figura.

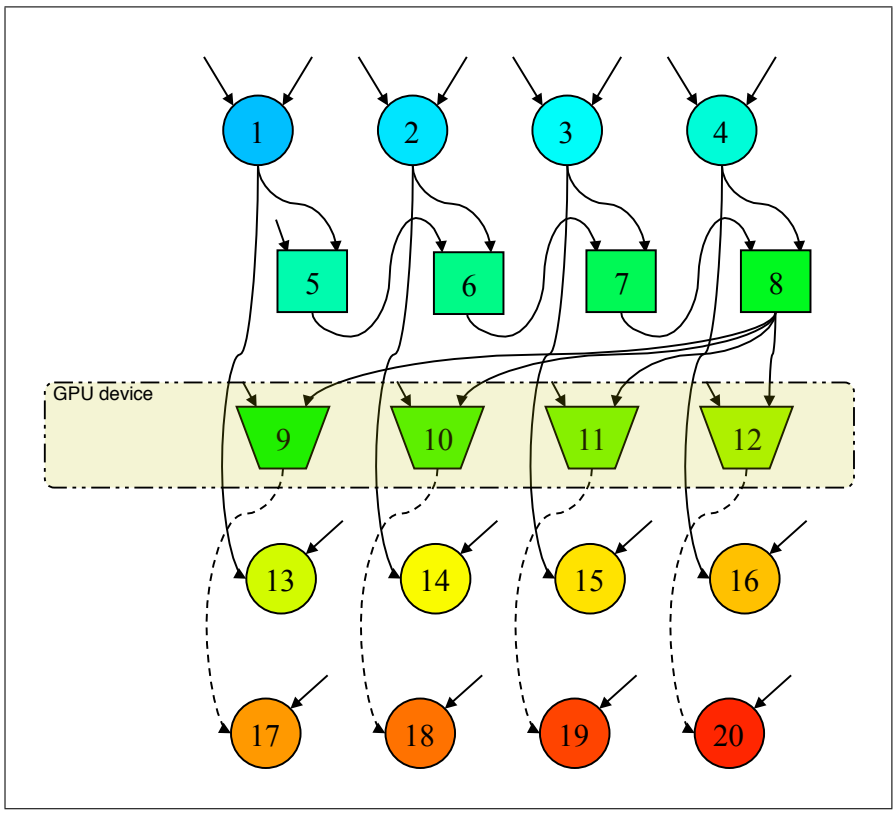

Figura 2.1: Ejemplo *OmpSs*: secuencia de ejecución de las tareas. Las circunferencias, trapecios y cuadrados son las funciones **vadd3**, **scale_add** y **accum** respectivamente. La numeración indica la secuencia de ejecución.

Capítulo 3

OpenCL

OpenCL nació de la necesidad de unificar el uso de las diferentes unidades de computación. El estándar pretende aprovechar estas unidades de computación y obtener la misma funcionalidad y rendimiento que utilizando *API*s específicas de los fabricantes, como puede ser *CUDA* para las *GPU nVidia*[2]. Para ello, el estándar define una arquitectura heterogénea que permite especificar el modo de uso de cualquier dispositivo.

En la figura 3.1 se detalla la arquitectura básica que sigue *OpenCL*. La aplicación (capa de aplicación en la figura) debe utilizar una *API* para acceder al *framework* de *OpenCL*. Los fragmentos/funciones que se quieran acelerar en un dispositivo deberán escribirse en lenguaje *C* de *OpenCL*. Estos fragmentos se les denomina *OpenCL kernels*.

Los *OpenCL kernels* están escritos en una extensión del estándar *C99* que permite explotar el paralelismo. Para que se puedan ejecutar dentro de un dispositivo es necesario compilar el código utilizando la *API* de *OpenCL*, tal y como muestra la figura 3.2.

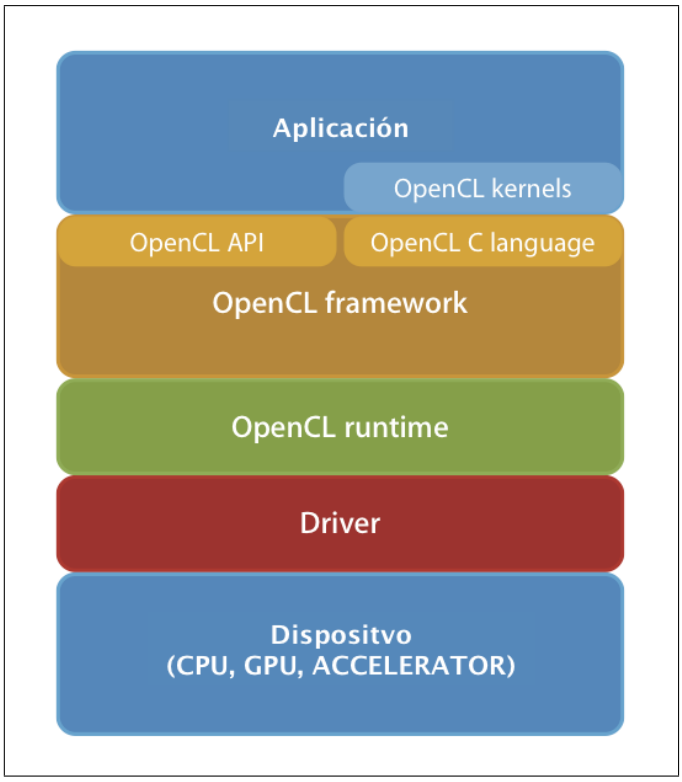

Figura 3.1: Arquitectura básica de *OpenCL*

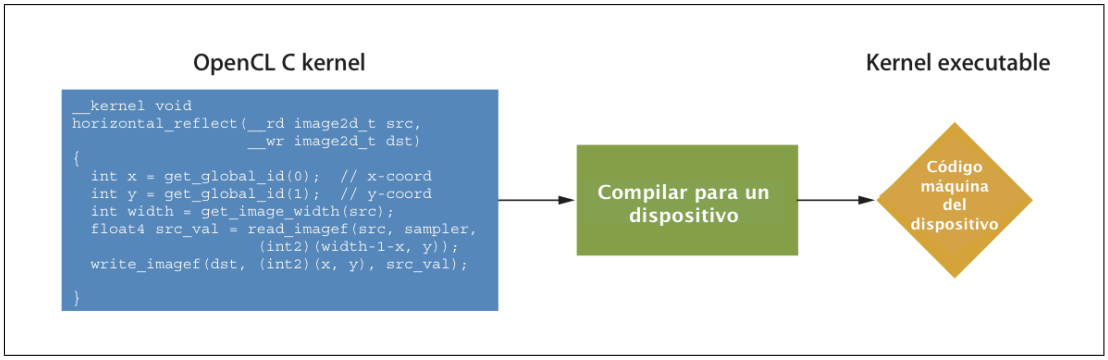

Figura 3.2: Código C de *OpenCL* a código del dispositivo

3.1. API de OpenCL

Para entender como funciona la *API* de *OpenCL* explicaremos primero los objetos principales que la componen:

- **Plataform**: Representa la implementación de un *runtime* de *OpenCL*. A partir de ésta se crean todos los objetos necesarios para hacer funcionar los dispositivos asociados a dicho *runtime*.
- **DeviceID**: Identifica un dispositivo con soporte de *OpenCL*.
- **Context**: Es el espacio de trabajo donde se instanciarán los objetos/dispositivos que vayamos a utilizar. Se inicializa con una *platform* y un conjunto de *DeviceID*'s.
- **Program**: Representa el código *C* de *OpenCL* a ejecutar dentro de un dispositivo. El *program* está siempre situado dentro de un contexto y tiene asociado un subconjunto de dispositivos inicializados dentro de dicho contexto.
- **MemObject**: Representa las regiones de memoria que se encuentran dentro de un contexto.
- **Kernel**: Es una instancia de un programa que servirá para ejecutarlo. Para ello se asociarán objetos de memoria para la entrada y salida de datos al dispositivo.
- **Event**: Representa el estado de una operación y permiten sincronizarnos con las operaciones asícronas de *OpenCL*.
- **CommandQueue**: Permite ejecutar los *kernels*, realizar operaciones de lectura y escritura de datos en los objetos de memoria, devolviéndonos eventos asociados a dichas operaciones.

Para una mejor comprensión, en la figura 3.3 mostramos cuál es la relación entre estos objetos de *OpenCL*. A continuación, detallamos los métodos más relevantes de la *API* de *OpenCL* que ayudan al desarrollo del proyecto.

- **clGetPlatformIDs**: Obtiene las implementaciones del *runtime* disponibles del sistema.
- **clGetDeviceIDs**: Obtiene la lista de dispositivos disponibles de dentro la *platform*.
- **clCreateContext**: Crea un entorno de trabajo situado dentro una *plataform* y un subconjunto de dispositivos de dicha *platform*.
- **clCreateCommandQueue**: Crea una *CommandQueue* dentro de un contexto y para un dispositivo.
- **clCreateBuffer**: Crea un objeto de memoria dentro del *context* indicado.
- **clEnqueReadBuffer**: Solicita una operación de lectura de datos de un objeto de memoria utilizado por el dispositvo. Esta operación precisa de una *CommandQueue*.
- **clEnqueWriteBuffer**: Solicita una operación de escritura de datos de un objeto de memoria utilizado por el dispositvo. Esta operación precisa de una *CommandQueue*.
- **clCreateProgramWithSource**: Crea un *program* a partir del código fuente escrito en lenguaje *C* de *OpenCL*.
- **clBuildProgram**: Compila el código para el conjunto de dispositivos indicado.
- **clCreateKernel**: Crea un *kernel* a partir de un *program*.
- **clSetKernelArg**: Inicializa un argumento del *kernel* con un objeto de memoria si éste es un puntero. En caso de no ser un puntero, simplemente hay que copiar los datos.
- **clEnqueueTask**: Solicita a una *CommandQueue* una operación de ejecución del *kernel*.

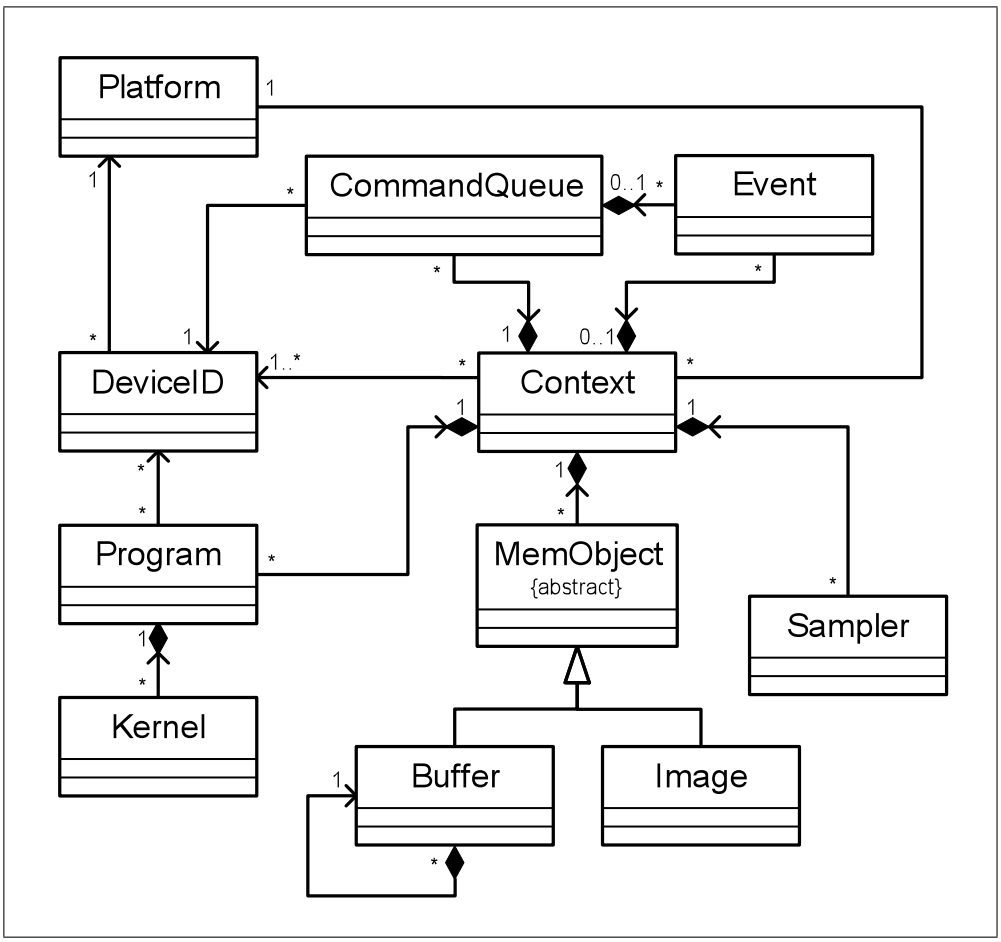

Figura 3.3: Diagrama de clases UML de *OpenCL* 1.1.

- **clWaitForEvents**: Espera hasta que las operaciones pendientes, representado por los *events* que le pasen como argumentos, hayan concluido.
- **clFinish**: Espera hasta que *CommandQueue* haya terminado de ejecutar todas las operaciones pendientes.

En la figura 3.4 podemos ver cómo se utilizan algunos métodos en la interacción entre los objetos de *OpenCL*.

El código 3.1 muestra un ejemplo detallado del uso de la *API* de *OpenCL* en C. El cuerpo de la función *KernelSource* corresponde con el código a acelerar. El código de la función *main* realiza toda la gestión de inicialización, compilación de códigos, transferencia de datos, ejecución del *kernel*, sincronización, etc.

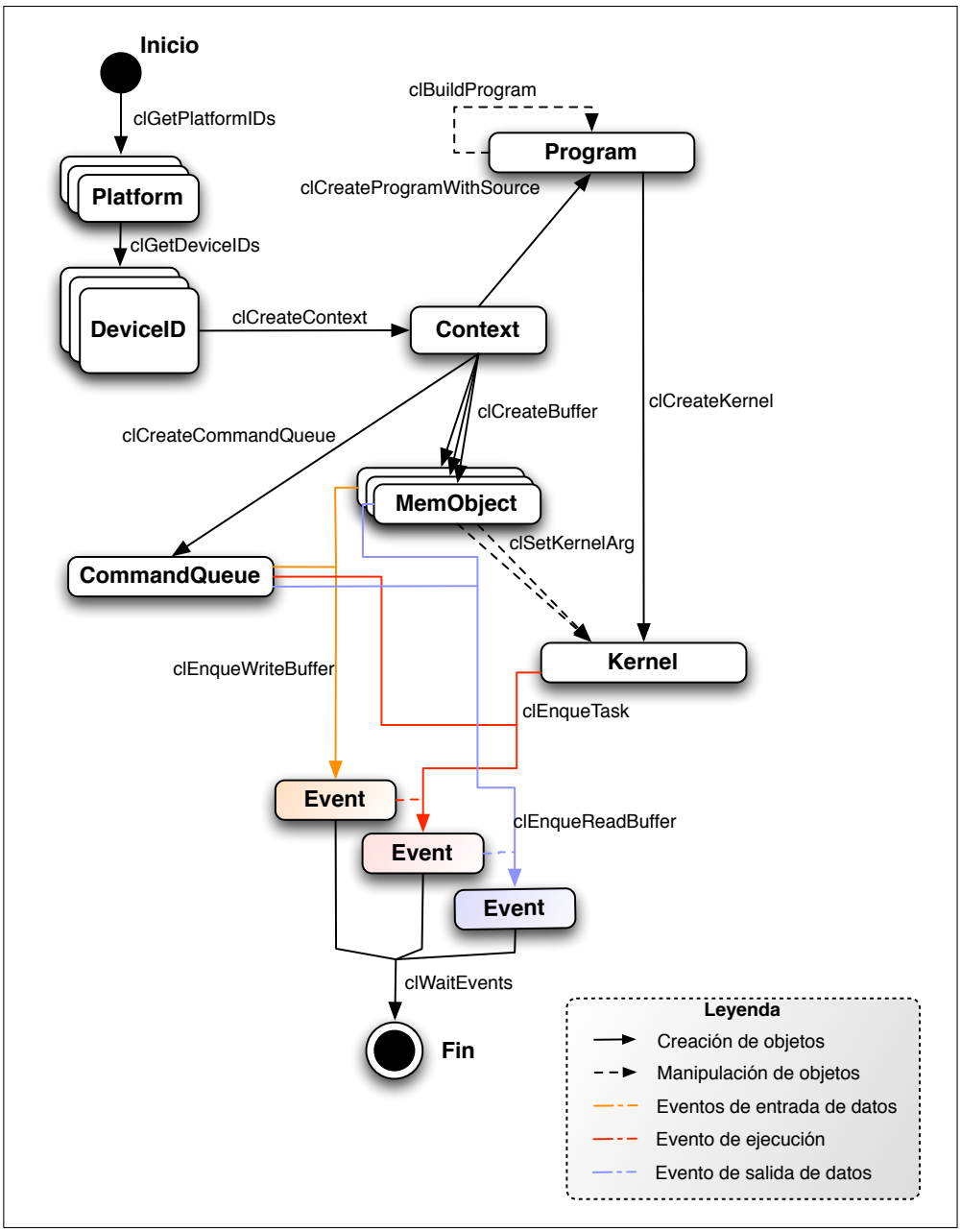

Figura 3.4: Diagrama de ejemplo del uso de *OpenCL*.

```
1 #include <CL/opencl.h>
2 ...
3 #define DATA_SIZE (1024)
4
5 const char *KernelSource =
6     "__kernel void square(                          \n" \
7     "    __global float* input,                     \n" \
8     "    __global float* output,                    \n" \
9     "    const unsigned int numOfElements)          \n" \
10    "{                                              \n" \
11    "    unsigned int i;                            \n" \
12    "    for(i=0;i<numOfElements;i++)               \n" \
13    "       output[i] = input[i] * input[i];         \n" \
14    "}                                              \n" \
15    "\n";
16
17 int main(int argc, char** argv) {
18    ...
19    float data[DATA_SIZE];          // Datos de entrada
20    float results[DATA_SIZE];       // Resultado de la operacion
21
22    cl_platform_id platform_id;     // Plataform
23    cl_device_id device_id;         // DeviceID
24    cl_context context;             // Context
25    cl_command_queue queue;         // CommandQueue
26    cl_program program;             // Program
27    cl_kernel kernel;               // Kernel
28    cl_mem input;                   // MemObject: Region de memroia
          para la entrada datos en el dispositivo
29    cl_mem output;                  // MemObject: Region de memroia
          para la salida de datos en el dispositivo
30    cl_event events[3];             // Events
31
32    // Inicializamos los datos como ejemplo.
33    ...
34
35    // Obtenemos algun runtime de OpenCL disponible
36    clGetPlatformIDs(1,&platform_id,&t);
37
38    // Obtenemos 1 dispositivo dentro de la platform
39    clGetDeviceIDs(platform_id,CL_DEVICE_TYPE_ALL,1,&device_id,NULL
          );
40
41    // Creamos un contexto
42    cl_context_properties properties[] = {CL_CONTEXT_PLATFORM,(
          cl_context_properties) platform_id,0};
43    context = clCreateContext(properties,1,&device_id,NULL,NULL,
          NULL);
44
45    // Inicializamos una commandQueue para el dispositivo
46    queue = clCreateCommandQueue(context,device_id,0,NULL);
47
48    // Creamos un Program que contega el codigo a accelerar
```

```
49    program = clCreateProgramWithSource(context,1,(const char **) &
          KernelSource,NULL,NULL);
50
51    // Compilamos el codigo para el dispositivo
52    errorCode = clBuildProgram(program,0,NULL,NULL,NULL,NULL);
53
54    // Creamos un MemObject para la entrada de datos
55    input = clCreateBuffer(context,CL_MEM_READ_ONLY,sizeof(float) *
          DATA_SIZE,NULL,NULL);
56
57    // Creamos un MemObject para la salida de datos
58    output = clCreateBuffer(context,CL_MEM_WRITE_ONLY,sizeof(float)
          * DATA_SIZE,NULL,NULL);
59
60    // Inicializamos los datos de entrada con una operacion de
          escritura al dispositivo
61    clEnqueueWriteBuffer(queue,input,CL_FALSE,0,sizeof(float) *
          DATA_SIZE,data,0,NULL,&events[0]);
62
63    // Creamos un kernel con el programa
64    kernel = clCreateKernel(program,"square",NULL);
65
66    // Inicializamos los argumentos del kernel
67    clSetKernelArg(kernel,0,sizeof(cl_mem),&input);
68
69    clSetKernelArg(kernel,1,sizeof(cl_mem),&output);
70    i = DATA_SIZE;
71    clSetKernelArg(kernel,2,sizeof(unsigned int),&i);
72
73    // Solicitamos la ejecucion del kernel
74    clEnqueueTask(queue,kernel,1,&events[0],&events[1]);
75
76    // Solicitamos leer el resultado
77    clEnqueueReadBuffer(queue,output,CL_FALSE,0,sizeof(float) *
          DATA_SIZE,results,2,&events[0],&events[2]);
78
79    // Esperamos hasta que haya finalizado todas las operaciones
80    clWaitForEvents(3,events);
81
82    // Liberamos memoria
83    clReleaseEvent(events[0]);
84    clReleaseEvent(events[1]);
85    clReleaseEvent(events[2]);
86    clReleaseKernel(kernel);
87    clReleaseMemObject(input);
88    clReleaseMemObject(output);
89    clReleaseProgram(program);
90    clReleaseCommandQueue(queue);
91    clReleaseContext(context);
92    ...
93 }
```

Código 3.1: Ejemplo del uso de OpenCL

Capítulo 4

Mercurium

Mercurium es un compilador *source-to-source* con soporte para los lenguajes de programación *Fortran*, *C* y *C++*. Éste es un compilador ágil, donde se puede transformar código de lenguaje en alto nivel a alto nivel de una manera sencilla. *Mercurium* no obtiene código ejecutable y, por consiguiente, será necesario el uso de otro compilador para acabar el proceso de compilación hasta obtener código binario para una arquitectura concreta.

El funcionamiento de *Mercurium* consta de varias etapas. La figura 4.1 muestra estas etapas. Primero de todo se realiza un análisis léxico del código de entrada (*Analizador C/C++* en la figura). En este proceso también se realiza una transformación a una representación intermedia. Una vez llegados a este punto, se pasa a la transformación de la representación intermedia según los parámetros de compilación utilizados y las directivas de transformación o ejecución que hayan en el código. Este proceso se realiza en varias fases tal como se muestra en la figura. En la última fase se acaba transformando esta representación intermedia a código *C*. Finalmente, el código resultante debe pasarse por otro compilador para poder obtener el código binario ejecutable (*Compilador final* y *Enlazador* de la figura).

Para poder automatizar todas estas etapas, la infraestructura de compilación de *Mercurium* tiene la posibilidad de configurar qué etapas se deben realizar. En la

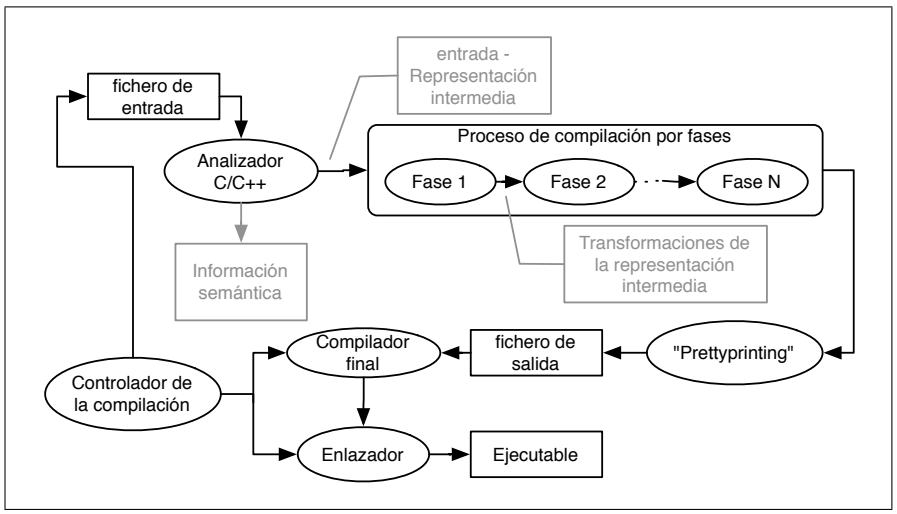

Figura 4.1: Proceso de compilación de *Mercurium*

figura 4.1, el *Controlador de la compilación*, es el que controla todas las etapas en función de dicha configuración.

Capítulo 5

Nanos

Nanos es un entorno de ejecución que ofrece una infraestructura para la investigación de modelos de programación en paralelo con acceso a memorias compartidas y procesadores con varios núcleos.

Nanos esta formado principalmente por tres componentes:

(a) El compilador *Mercurium* que hemos explicado en el capitulo 4. En la figura 5.1 podemos ver el proceso que sigue *Mercurium* para generar un ejecutable con *Nanos*. Este proceso está dentro de una de las fases de transformación de código en representación intermedia, explicado anteriormente.

(b) El *Runtime* que encapsula las funcionalidades de ejecución y control de tareas en ejecución. Éste normalmente se implementa dentro de un conjunto de librerías dinámicas. La figura 5.2 muestra las componentes del *runtime* de *Nanos*.

(c) Las herramientas de instrumentación que permiten hacer un seguimiento y análisis de la ejecución de las aplicaciones.

Actualmente *Nanos* tiene soporte para diversos paradigmas de programación: *OpenMP*, *OmpSs*, *StarSs* y *Chapel*[8]. Para ello divide el proceso en dos partes: Por un lado utiliza el compilador *Mercurium* para interpretar y generar el código necesario para

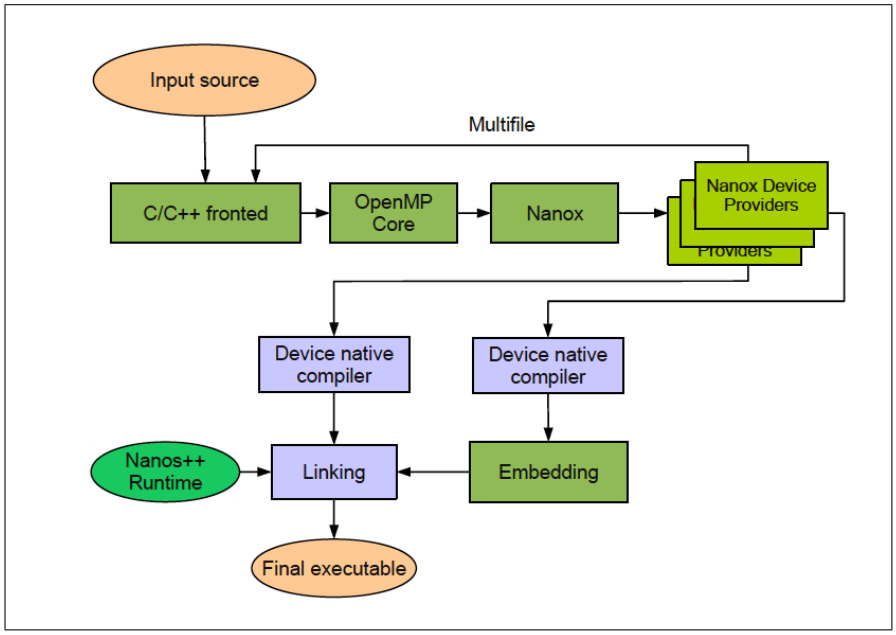

Figura 5.1: Proceso de *Mercurium* para generar un ejecutable con soporte de *Nanos*

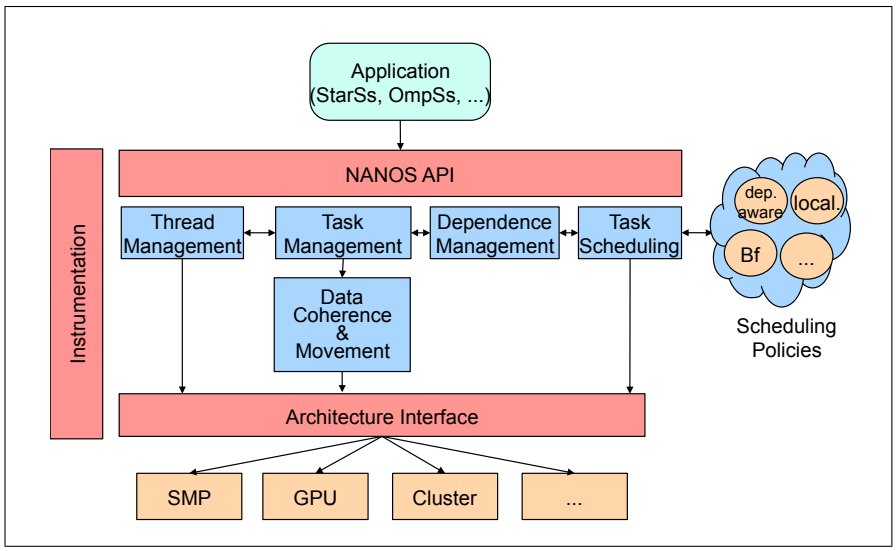

Figura 5.2: Componentes de *Nanos runtime* con instrumentación

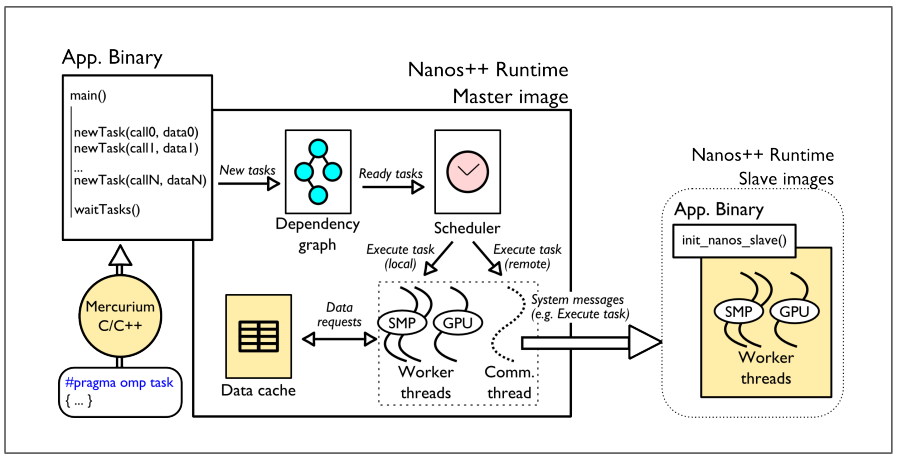

Figura 5.3: Esquema de compilación y ejecución de una aplicación sobre *Nanos*

hacer las llamadas al *runtime* del paradigma en cuestión y, por otro lado, implementa el *runtime* que da soporte a la gestión de los dispositivos que soporta el paradigma. La figura 5.3 muestra un esquema de las etapas de compilación y ejecución de una aplicación que utiliza *Nanos* como infraestructura de compilación y ejecución.

Capítulo 6

OmpSs$_{CL}$: OmpSs + OpenCL Framework

6.1. Descripción y funcionamiento

OmpSs$_{CL}$ es el entorno de compilación y ejecución que hemos desarrollado para ofrecer las funcionalidades de *OmpSs* sobre el estándar *OpenCL*. Para llevar a cabo el proyecto hemos utilizado la infraestructura de *Nanos*. El entorno de ejecución para el que se ha desarrollado es una plataforma con un procesador *host*, con uno o más núcleos, que tiene uno o más dispositivos de computación asociados.

Nuestro sistema de compilación y ejecución está preparado para soportar todas las funcionalidades que la especificación de *OpenCL* nos ofrece para la ejecución concurrente de tareas.

Aún así, queremos hacer notar que las implementaciones actuales de *OpenCL* carecen de la funcionalidad de ejecución concurrente en un mismo dispositivo.

El proyecto se divide en dos partes: *a*) Un *compilador source-to-source* utilizando la infraestructura de *Mercurium* y *b*) el *runtime* encapsulado dentro de una librería dinámica. La Figura 6.1 muestra el esquema del funcionamiento de la infraestructura de compilación y ejecución *OmpSs*$_{CL}$. El compilador analiza el código de usuario y utiliza la información de las construcciones y cláusulas para generar un nuevo código transformado. Este código transformado contiene las llamadas a nuestro *runtime* para poder controlar los recursos de los dispositivos por medio del *OpenCL*. Además, el nuevo código incluye nuevas construcciones y cláusulas que ayudarán al compilador, en siguientes pasos, a generar las llamadas necesarias al *runtime* de *Nanos* para controlar la dependencias. Una vez generado el nuevo código en lenguaje *C* con llamadas al *runtime*, el controlador de *Mercurium* compila ese código y obtiene el ejecutable de la aplicación para el procesador *host*.

Nuestro compilador también genera un archivo para cada *kernel* que se tiene que acelerar. Estos archivos están en lenguaje *C* de *OpenCL*. Recordemos que *OpenCL* necesita las fuentes de los códigos en tiempo de ejecución, y es en ese momento que los compila para los dispositivos especificados.

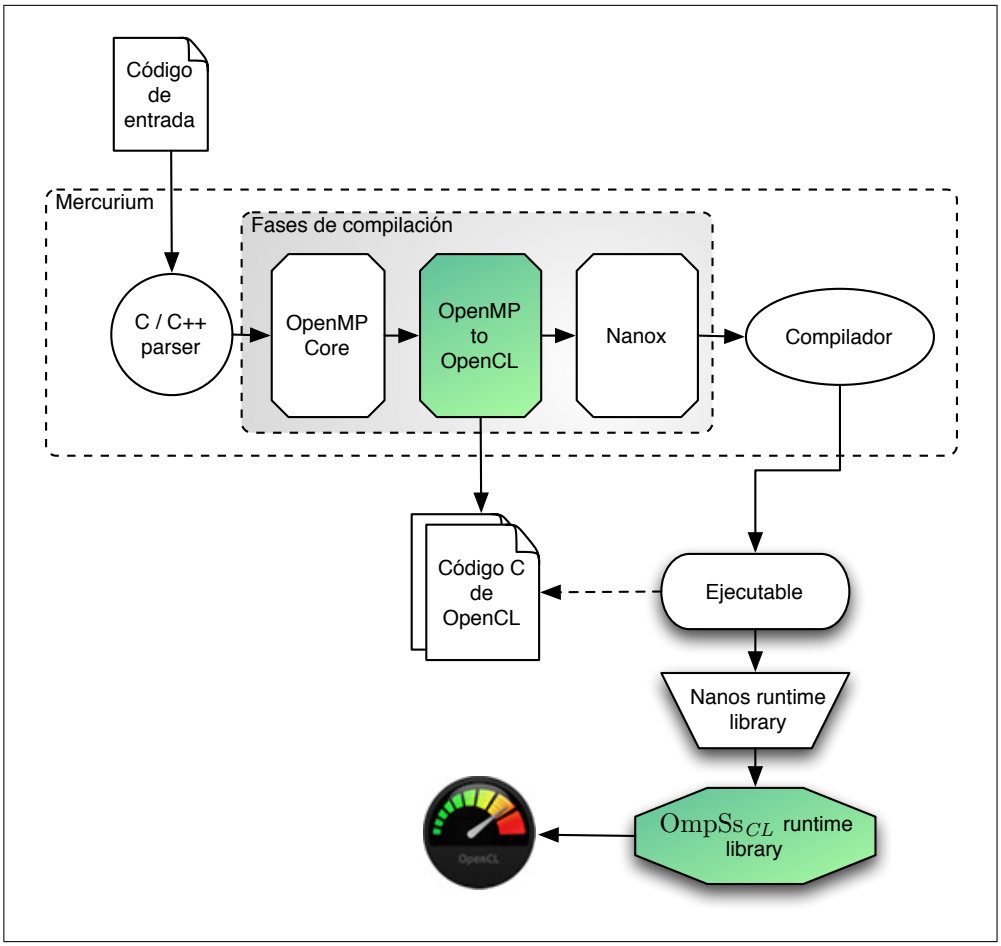

Figura 6.1: Esquema del funcionamiento del *OmpSs*$_{CL}$

En el momento que ejecutemos el programa se cargarán las librerías del *runtime* de *Nanos*, las desarrolladas en el proyecto para la gestión de recursos de los dispositivos y las del *OpenCL*. La ejecución resultante será un código que utilizará *Nanos* para generar las tareas. Así, mientras que *Nanos* se encarga de controlar las dependencias entre estas tareas y planifica cómo se ejecutan, nuestro *runtime* efectuará las transferencias de memoria necesarias, cargará los códigos de *C* de *OpenCL*, y hará un control de recursos para el correcto uso de la especificación de *OpenCL*.

6.2. Instalación y uso

6.2.1. Requisitos del sistema

Sería recomendable disponer de varios dispositivos con soporte de *OpenCL*. Los dispositivos que soporta *OpenCL* son las *GPU*s, las *CPU*s con soporte para instrucciones *SSE2*[1][7], y el *Cell BE*.

Para utilizar nuestro sistema de compilación, será necesario cumplir primero con los requisitos de instalación de *Mercurium* y de *Nanos*. A continuación detallamos la lista completa de requisitos del sistema:

- Entorno GNU Linux
- GNU flex 2.5.4 (o superior)
- GNU gperf 3.0.0 (o superior)
- GNU bison 2.3 (o superior)
- automake-1.10 (o superior)
- autoconf-2.63 (o superior)
- libtool-2.2.6a (o superior)
- GNU gcc con soporte C++ (4.1 o superior)
- git (si se desea descargar *Nanos* y *Mercurium* del repositorio)
- Controladores de los dispositivos con soporte de *OpenCL*.
- *SDK* de *OpenCL*.
- grpahviz y doxygen (si se desea generar la documentación de *Nanos* y *Mercurium*)

[1] Streaming SIMD Extensions 2. Es un conjunto de instrucciones de la arquitectura IA-32 SIMD.

- libxml2-dev (si se desea instrumentar el código de *Nanos*)
- Mercurium 1.3.5.7 (o superior, obligatoriamente con el código fuente)
- Nanox 0.6a (o superior, obligatoriamente con el código fuente)
- Extrae 2.1.1 y Paraver (si queremos instrumentación)

6.2.2. Instalación

El primer paso es descargar el software necesario con los paquetes disponibles del repositorio del entorno Linux. Además, si carecemos de una instalación previa de *Nanos*, *Mercurium*, *Extrae* y *Paraver* podemos utilizar las siguientes instrucciones para descargar la última versión disponible del repositorio online:

```
1 mkdir "source-downloads"; cd "source-downloads"
2 git clone "http://pm.bsc.es/git/mcxx.git"
3 git clone "http://pm.bsc.es/git/nanox.git"
4 # Si queremos las herramientas de instrumentacion:
5 wget "http://www.bsc.es/ssl/apps/performanceTools/files/extrae
    -2.1.1-p2.tar.gz"
6 wget "http://www.bsc.es/ssl/apps/performanceTools/files/
    wxparaver32.tar.gz"
7 cd ..
```

Si además se quiere instrumentar el código, deberemos bajarnos una versión de *Extrae* (librería de instrumentación) ya compilada con soporte para *Nanos*. En caso contrario, podemos ejecutar las siguientes instrucciones para instalarla:

```
1 mkdir build;
2 tar -xzvf "source-downloads/extrae-2.1.1-p2.tar.gz" -C build/
3 mkdir build/extrae; cd build/extrae
4 %PREFIX sera la ruta donde queremos instalarlo.
```

```
./configure --prefix="$PREFIX" --enable-nanos --enable-pthread
make; make install;
cd ../..
%En caso de que no tengamos el Paraver para visualizar los
    resultados de la instrumentacion
tar -xzvf "source-downloads/wxparaver32.tar.gz" -C /tmp
cp -r /tmp/$PARAVER_NAME_FILE/* -t install2
rm -f -r "/tmp/$PARAVER_NAME_FILE"
```

Llegados a este punto, ya podemos compilar *Nanos*. Sin embargo, por incompatibilidades con la implementación de *OpenCL* de *ATI*, se deben realizar algunas modificaciones al código de gestión de memoria de *Nanos*. En particular, debemos desactivar parte del código que redefine los operadores *new* y *delete*. Eso se puede realizar automáticamente con los siguientes comandos:

```
cd source-downloads/nanox/
cp src/support/new.cpp src/support/new.cpp.backup
sed -i "s|^void|//void|g" src/support/new.cpp
```

Por otra parte, si queremos compilar *Nanos* con soporte para la instrumentación con *Extrae*, también tendremos que modificar otro archivo. Esto es debido a que los proyectos están en constante evolución, y *Nanos* está usando una *API* anterior a las versiones actuales del *Extrae*. A continuación mostramos las líneas de comandos que nos ayudan a solucionar este problema:

```
cp src/plugins/instrumentation/extrae.cpp src/plugins/
    instrumentation/extrae.cpp.backup
sed -i "s|ce.Values = (unsigned int \*) alloca (ce.nEvents \*
    sizeof (unsigned int));|ce.Values = (unsigned long long *)
    alloca (ce.nEvents * sizeof (unsigned long long));|g" src/
    plugins/instrumentation/extrae.cpp
```

```
sed -i "s|for ( int|for ( unsigned int|g" src/plugins/
    instrumentation/extrae.cpp
```

A partir de aquí ya podemos compilar e instalar *Nanos*. Los siguientes comandos ayudan a realizar este proceso:

```
cd "/source-downloads/nanox";
autoreconf --force --install
cd ../..
mkdir build/nanos; cd build/nanos
../../source-downloads/nanox/configure \
   --prefix="$PREFIX" \
   --disable-gpu-arch \
   # PREFIX_INSTALLED_EXTRAE sera la ruta donde tenemos instalado
       el Extrae
   --with-extrae="$PREFIX_INSTALLED_EXTRAE"
make; make install;
```

En estos comandos podemos observar que hemos utilizado la desactivación de la *GPU*(opción *–disable-gpu-arch*). Esto se hace así porque *Nanos* tiene soporte para *GPU* por defecto, y éste acaba afectando al uso del *OpenCL*.

Finalmente, podemos compilar e instalar *Mercurium* con los siguientes comandos:

```
cd "source-downloads/mcxx"
autoreconf --force --install
cd ../..
mkdir build/mcxx; cd build/mcxx
../../source-downloads/mcxx/configure" \
   --prefix="$PREFIX" \
   --enable-tl-openmp-nanox \
   --enable-ompss \
   --with-nanox="$PREFIX_INSTALLED_NANOS" \
```

```
10     --enable-tl-superscalar \
11     --with-superscalar-runtime-api-version=5
12 make; make install;
```

Todo este proceso de instalación del *Mercurium*, *Nanos*, *Extrae* y *Paraver* puede llegar a ser tedioso. Para facilitarlo se ha creado un *script* que automatiza y controla todos los pasos. Este*script* esta incluido en la distribución del sistema creado.

Una vez se tienen instalados todos los paquetes, ya podemos instalar nuestro sistema. Para instalar nuestro proyecto hace falta saber donde se encuentra *Mercurium*. El camino hasta *Nanos* ya está incluido en la configuración del *Mercurium*. Por defecto, el proyecto está configurado para encontrar *Mercurium* dentro del directorio local. En otro caso, el directorio se puede configurar dentro del archivo *mcxxCLN/Makefile* del proyecto.

```
3 MERCURIUM_SOURCE_DIRECTORY=./../tools/source-downloads/mcxx
4 MERCURIUM_BUILD_DIRECTORY=./../tools/build/mcxx
5 MERCURIUM_INSTALL_DIRECTORY=./../tools/install2
```

También es necesario definir la ruta donde tenemos instalado el *SDK* de *OpenCL*. Esto se puede hacer en el archivo OpenCLNanos/Makefile del proyecto.

Por otra parte, si queremos instrumentación deberemos poner la ruta de *Nanos* ya que dentro de nuestro *runtime* se deberá acceder a la *API* de *Nanos* para añadir eventos de instrumentación.

```
3 OPEN_CL_INC=/opt/AMD-APP-SDK-v2.4-lnx32/include
4 OPEN_CL_LIB=/opt/AMD-APP-SDK-v2.4-lnx32/lib/x86
5
6 TOOLS_INSTALL_PATH=../tools/install2
7 NANOS_INC=$(TOOLS_INSTALL_PATH)/include/nanox
```

```
30 # Agregar "-I $(NANOS_INC) -DENABLE_INSTRUMENTATION_NANOS" si se
       quiere instrumentacion
31 CXXFLAGS = -Wall -fPIC -g -pthread
```

Para finalizar la instalación debemos compilar nuestra parte del compilador y *runtime*, integrándolo todo dentro de las fases de compilación del *Mercurium*

```
1 # Compilador
2 cd mcxxCLN;
3 make;
4 make install # Se instala dentro del Mercurium
5 make install-test-links # Sirve para crear accesos directos en la
      carpeta ./test de nuestro compilador
6
7 # Runtime
8 cd OpenCLNanos
9 make # No hay que instalar, solo dejamos la libreria dinamica
```

En la figura 6.2 vemos el proceso necesario que hacemos para integrar el proyecto dentro de *Nanos* y *Mercurium*. Por un lado creamos los accesos directos, donde "ssclncc" es el compilador para lenguaje *C* y "ssclncxx" para *C++*. El archivo de configuración lo generamos a partir de la configuración original del entorno *Nanos*, llamado "config.ssc". Nótese que *Nanos* es un proyecto complejo, y es por ello que para facilitar esta tarea, hemos automatizado también esta tarea. El que lo automatiza crea una nueva configuración, renombrando algunas reglas y añadiendo la localización de nuestra fase de compilación y del *runtime* para crear el ejecutable final.

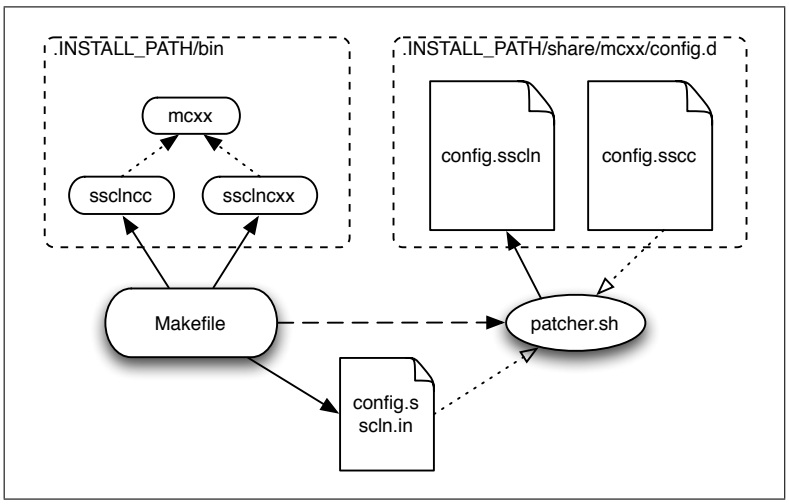

Figura 6.2: Integración del proyecto dentro de *Nanos* y *Mercurium*

6.2.3. Uso

Para usar $OmpSs_{CL}$ simplemente debemos invocarlo pasándole el código fuente de la aplicación a compilar. El binario del compilador se encuentra dentro la instalación de *Mercurium*. En concreto, en la carpeta de la instalación se han creado accesos directos por defecto.

```
1 ssclncc codigoEjemplo.c -o ejemplo
```

Con este comando obtendremos el ejecutable de la aplicación para el procesador *host*, y un conjunto de archivos que representarán las funciones que han sido convertidas en códigos en lenguaje *C* de *OpenCL* para ejecutarse dentro del los dispositivos. Los nombres de estos archivos son de la forma: <nombre del código de entrada>_<nombre de la función>_ssclncc.cl. Una vez tenemos compilado el programa, ya podemos ejecutar el archivo binario indicando el número de hilos de ejecución deseado. Esto se hace definiendo el valor de la variable de entorno *OMP_NUM_THREADS*. En el siguiente comando se hace justo en el momento de lanzar a ejecutar el programa *ejemplo*:

```
1 OMP_NUM_THREADS=4 ./ejemplo [argumentos del ejemplo]
```

En el caso de querer hacer una ejecución usando la instrumentación, deberemos haber compilado el código con *–instrumentation*:

```
1 ssclncc codigoEjemplo.c -o ejemplo --instrumentation
```

y luego ejecutarlo con opciones de instrumentación activadas, por ejemplo:

```
1 NX_ARGS=' --keep-mpits --instrumentation=extrae ' OMP_NUM_THREADS=4
      ./ejemplo [argumentos del ejemplo]
```

Finalmente, en el momento de compilar con nuestro compilador podemos indicar algunas opciones:

- **- - disableNanos**: deshabilita la utilización de *Nanos*. Con esta opción se ejecutará el código secuencialmente. Nótese que es *Nanos* el que nos facilita la tarea de creación de hilos de ejecución, planificación y el soporte de control de dependencias entre las tareas. Además, esta opción, combinada con la opción **-y** , hará que el compilador genere el/los archivos intermedios en el proceso de compilación. Estos códigos pueden usarse para posteriormente compilarlos activando *Nanos* o bien otras implementaciones de *OmpSs*. Estos ficheros intermedios son los códigos *C* originales transformados.
- **- - variable=generate_report_on_taskwait:1**: permite generar un pequeño informe con los tiempos de utilización de los dispositivos de *OpenCL* para cada construcción *taskwait*
- **- - variable=delete_original_functions:value**: donde el *value* puede ser 0,1 o 2. Esta opción permite al compilador eliminar el cuerpo original de la función asignada a convertirse en tarea. Si el valor de esta opción es '1', el compilador detecta automáticamente si debe eliminar el cuerpo de la función ya que existen

*pragma*s del tipo *verbatim*[2] de *Mercurium* que encapsulan código que puede que no compilen para la arquitectura del procesador *host*. La opción '0' conserva los cuerpos de la función mientras que la opción '2' los elimina todos.

6.2.4. Requisitos de los códigos de entrada

El sistema $OmpSs_{CL}$ creado accepta las aceleración de funciones que tengan construcción *task*[6]. Cualquier otro códgio que no sea una función será tratada igual que lo haría *OmpSs* por defecto.

En concreto, los requisitos de entrada para las construcciones de los *pragma*s son:

- La construcción *target* debe ir acompañada de la cláusula *device* indicando los dispositivos en que se pueda ejecutar la *task*. Se aceptan dos tipos de dispositivo: El dispositivo *CPU* representado por las palabras: *cpu* o *smp* y otros dispositivos como aceleradores o *GPU*s representado por la palabra: *gpu*.

- Todas las transferencias de memoria a *OpenCL* deben estar explícitas. Estas se extrae de la información de dependencia de las construcciones *task*. Esto se ha realizado así porque en el momento de la implementación de nuestro compilador, las cláusulas *copy_deps, copy_in, copy_out, copy_inout* estaban en fase de depuración. El cambio de nuestra implementación para utilizar la información de los *copy_** no serían cambios significativos al funcionamiento de nuestro sistema.

- Se aceptan las cláusulas para indicar las dependencias en la construcción *task*. Dentro de estas cláusulas sólo se aceptan argumentos de la función que sean punteros, indicando el número de elementos con una *shaped expression*[3].

[2] Estas construcciones sirven para pasar código fuente directamente al dispositivo, sin que el compilador de *Mercurium* lo analice

[3] Consiste en poner delante del símbolo del puntero el número de elementos cerrado con corchetes. Por ejemplo sea *ptr* un puntero y *n* el número de elemento, la *shaped expression* sería "[*n*] *ptr*"

Si no queremos modificar los código de *C* de *OpenCL* generados, el cuerpo de las funciones con *pragma*s no deberán tener ninguna referencia a elementos externos de dicha función, es decir, no deberán utilizar variables globales ni tampoco otras funciones que no estén especificadas dentro del estándar[11].

En el código 6.1 podemos ver un ejemplo sencillo de código de entrada a la función *vector_add*, que se le han añadido los dos *pragma*s necesarios: el primero indicando que la función se puede ejecutar en cualquier dispositivo disponible y el segundo detallando la información para las dependencias de memoria. En el ejemplo vemos las variables *vs1* y *vs2* como entrada de datos y *vd* de salida. Además se esta indicando el número de elementos de estas variables con la *shaped expression*, en este caso es el valor de *BS*.

El código 6.2 es generado automáticamente a partir del cuerpo de la función. Este código lo genera nuestro compilador y es el que se ejecutará dentro de los dispositivos de *OpenCL*. En la sección 6.3.1 podemos encontrar más detalles sobre este proceso.

```
1  ...
2  const unsigned int BS = 1024;
3  ...
4  #pragma omp target device(smp,gpu) copy_deps
5  #pragma omp task input([BS] vs1, [BS] vs2) output([BS] vd)
6  void vector_add(int *vd, int *vs1,  int *vs2, int bs)
7  {
8     int i;
9     for (i=0; i<bs; i++)
10       vd[i] = vs1[i] + vs2[i];
11 }
12 ...
13 int main(int argc, char *argv[])
14 {
15    ...
16    vector_add(A,B,C,bs);
```

```
17     ...
18 }
```

Código 6.1: Ejemplo de utilización del OmpSs$_{CL}$

```
1 __kernel void vector_add(__global int *vd, __global int *vs1,
      __global int *vs2, __global int bs ) {
2     int i;
3     for (i = 0;
4          i < bs;
5          i++)
6          vd[i] = vs1[i] + vs2[i];
7 }
```

Código 6.2: Código C de OpenCL generado a partir del ejemplo de utilización del OmpSs$_{CL}$

6.3. Diseño e Implementación

En esta sección se explica: la generación de código en el proceso de compilación, la *API* para acceder a nuestro *runtime* y poder ejecutar código en *OpenCL*, y finalmente la gestión y optimización de los recursos de *OpenCL*.

6.3.1. Proceso de generación de código

Nuestro proceso de compilación fuente a fuente de generación de código se ejecutará después del análisis realizado por el *Mercurium* original, y está dividido en dos fases que se explican a continuación. En ellas deberemos generar las llamadas necesarias al *runtime* para poder utilizar *OpenCL*.

En la figura 6.3 podemos ver un ejemplo de la generación de código que nos ayudará a entender los pasos a realizar en estas dos fases. Los triángulos representan los lugares donde se insertará el código de las cajas flotantes y las cruces rojas indican que el código original será eliminado.

En la primera fase tenemos la inicialización de las tareas en el *runtime* de *OpenCL*, parte de esta inicialización es la compilación de los *kernles*. Para no compilar cada *kernel* en cada invocación, se hace una vez al principio del programa principal, tal y como se observa en la figura 6.3, en el primer triángulo de la función *main*. También se deben añadir las cabeceras necesarias al inicio del programa como vemos en el triángulo superior de la figura.

Para cada función con construcción se generará un fichero con código *C* de *OpenCL*, según la nomenclatura indicada en la sección 6.2.3. En la figura para el procedimiento *vector_add* se generará un archivo con nombre "<nombre del código de entrada>_vector_add_ssclncc.cl".

Finalmente, se hace la transformación de aquellas llamadas a funciones que se quieren acelerar. La transformación consiste en: *a*) Preparar el código con las construcciones que permitan crear las tareas con el *runtime* de *Nanos*, para que éste gestione

las dependencias. *b*) Gestionar los recursos de *OpenCL* con el código que se añade para hacer llamadas a nuestro *runtime*. Eso se puede ver en el código ensertado en substitución de las llamadas a *vector_add*.

En la segunda fase, el compilador analizará el código generado de la primera fase, con tal de generar las llamadas al *runtime* de *Nanos* que hará la gestión de dependencias. Este código final no se muestra en la figura.

En cuanto a los ficheros generados para cada *kernel*, la figura 6.4 muestra el código *C* de *OpenCL* para la función *vector_add*. Los principales cambios a la función original son: añadir "__kernel" a la definición de la función y añadir "__global" a cada uno de los argumentos de la función.

6.3.2. API de nuestro runtime

En esta sección se describen los procedimientos disponibles dentro de nuestro *runtime* para la gestión de los dispositivos y sus recursos con *OpenCL*. Para obtener una mayor compatibilidad con el resto del sistema de compilación hemos escrito la *API* de nuestro *runtime* en *C*. Para mayor detalle aconsejamos mirar el código fuente del proyecto.

Tipos y estructuras de datos:

cln_task_id : Tipo que identifica a un tarea.

cln_device_type : Tipo que representa un conjunto de dispositivos.

cln_arg_flags : Tipo que representa los tipos de argumentos, que pueden ser un puntero de entrada, salida, entrada y salida o tipo escalar.

cln_task_instance_info : Estructura de datos que encapsula la información necesaria para identificar una tarea y el conjunto de dispositivos donde se quiere ejecutar.

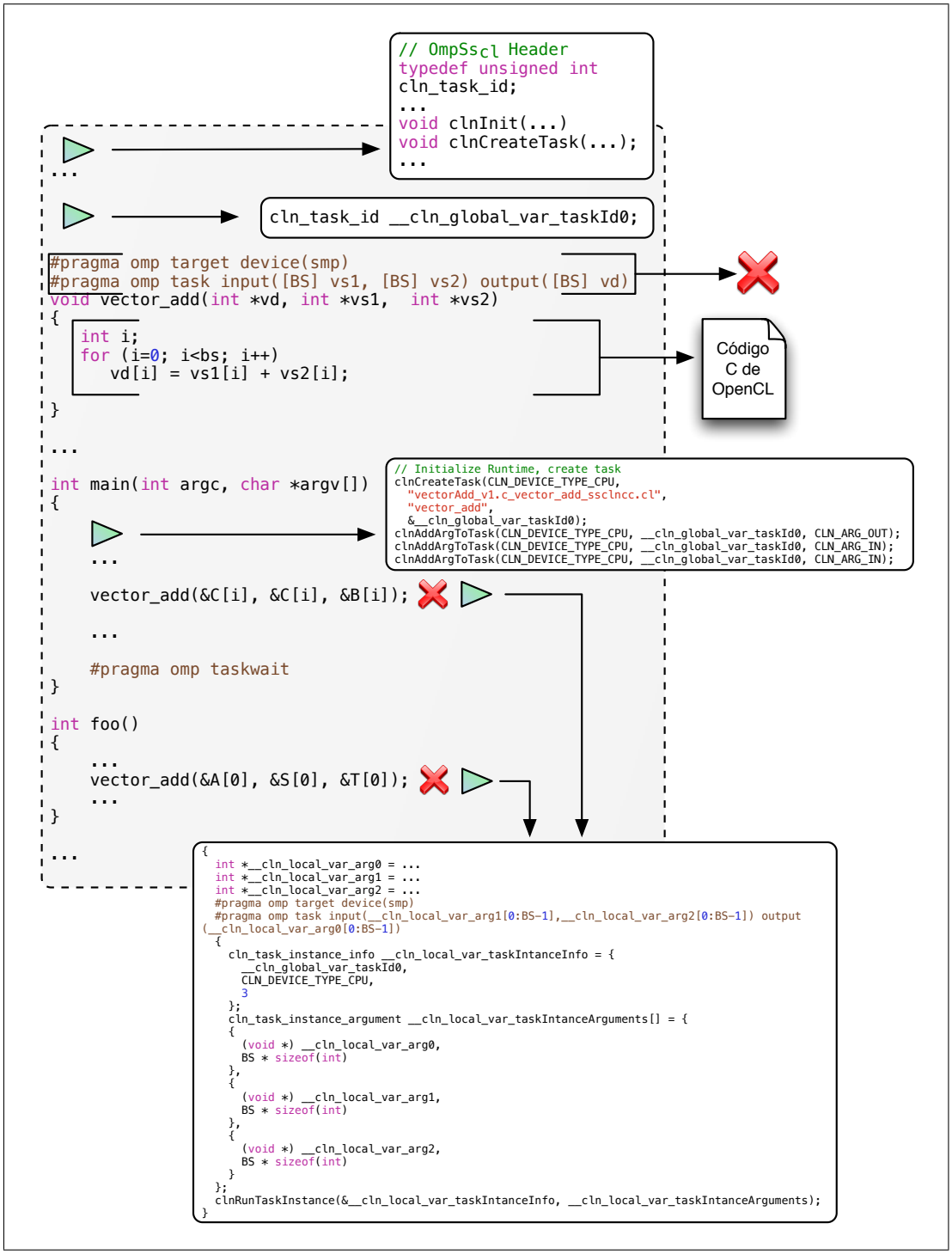

Figura 6.3: Ejemplo de la generación de código

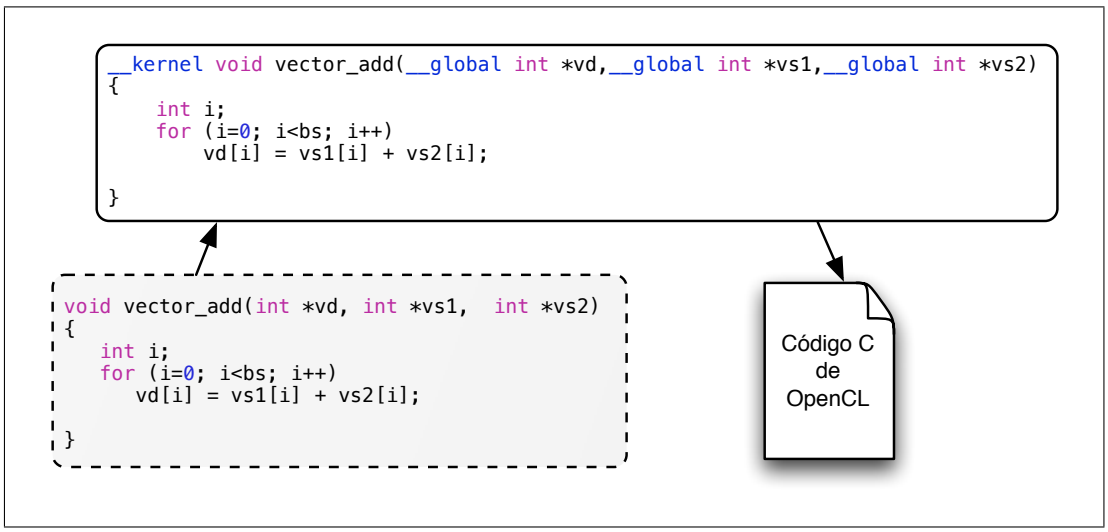

Figura 6.4: Generación de código *C* de *OpenCL* de la figura 6.3

cln_task_instance_argument : Estructura de datos que encapsula un puntero y su tamaño.

Procedimientos:

clnInit : Inicializa nuestro *runtime*, indicando los tipos de dispositivos que se van a utilizar. Esta llamada es opcional, ya que el *runtime* se puede inicializar automáticamente al crear un tarea.

clnCreateTask : Crea una tarea. A este procedimiento se le pasan los dispositivos donde se puede ejecutar la tarea, el nombre del archivo donde pueda encontrar el código a ejecutar dentro del dispositivo de *OpenCL* y el nombre de la función a ejecutar. Como argumento de salida se le pasa el identificador de la tarea.

clnAddArgToTask : Configura un argumento de ejecución de una tarea. La tarea se identifica con un determinado identificador obtenido con el procedimiento *clnCreateTask*.

clnRunTaskInstance : Ejecuta una tarea, identificada por su identificador, usando las estructuras de datos *cln_task_instance_info* y *cln_task_instance_argument* explicadas anteriormente.

clnReportUsage : Escribe por pantalla la información de uso de los dispositivos de *OpenCL*.

6.3.3. Instrumentación de nuestro runtime

Para instrumentar nuestro proyecto, será necesario sincronizarse con *Nanos*. Recordemos que *Nanos* planificará las tareas según la información de dependencias de las tareas y entrará en nuestro *runtime*. *Nanos* nos permite utilizar una *API* para instrumentar el código que se esté ejecutando, basada en una librería de instrumentación llamada *Extrae*[1].

La instrumentación que hemos realizado consiste en mantener la información sobre el estado del *runtime*. Para ello se utilizan una serie de eventos, con valores predefinidos para indicar el estado de un evento en cuestión.

A continuación detallamos las funciones que utilizamos para instrumentar el código con *Nanos*:

nanos_instrument_register_key : Crea un evento, indicándole la descripción y una palabra clave para identificarlo. Esta información la guarda dentro de un diccionario. La función retorna un identificador de *Nanos* para trabajar con el evento.

nanos_instrument_register_value_with_val : Registra información sobre el significado de un valor que puede tomar un evento en concreto.

nanos_instrument_enter_burst : Añade un valor para indicar el estado de un evento.

nanos_instrument_leave_burst : Elimina el último valor añadido sobre el estado de un evento.

6.3.4. Gestión de los recursos de OpenCL

Nuestro *runtime* puede ser accedido simultáneamente por varios hilos de ejecución para la ejecución de las tareas. Esto implica que debemos tener un control de los recursos disponibles, para evitar bloquearnos por falta de recursos. Por otra parte, también tenemos que cumplir que las operaciones de ejecución de tareas del *runtime* sean *thread-safe*.

Para cada operación de ejecución de tareas en nuestro *runtime* hay tres fases:

Primera fase : Encolar la ejecución de la tarea dentro del *OpenCL* con las transferencias de memoria requeridas, tanto de entrada como de salida de datos.

Segunda fase : Esperar la finalización de la ejecución de la tarea, incluyendo la terminación de las transferencias de memoria.

Tercera fase : Liberar los recursos de *OpenCL* destinados a la ejecución de la tarea.

Desafortunada las operaciones de *OpenCL* para llevar a cabo su ejecución no son *thread-safe*. Esto implica que la primera y la tercera fase hay que gestionarlas adecuadamente para evitar que más de un hilo pueda estar trabajando simultáneamente con *OpenCL*.

Funcionamiento del gestor de recursos

La figura 6.5 muestra un esquema del funcionamiento del gestor de recursos sobre *OpenCL*. Como se puede observar hay dos semáforos: uno en la entrada al gestor de recursos del planificador en sí mismo y otro en la entrada a *OpenCL*. El primero de ellos sirve para gestionar los recursos disponibles. El segundo es para garantizar que sólo un hilo esté ejecutando operaciones sobre *OpenCL*.

El funcionamiento básico de una operación de ejecución de una tarea sobre *OpenCL* mostrado en la figura 6.5 es el siguiente:

a) Cada operación debe hacer un bloqueo sobre el semáforo del gestor de recursos.

b) Si la operación es de liberación de recursos, se desbloquea el semáforo y hacemos un bloqueo al semáforo de entrada a *OpenCL*. Si la operación es de solicitud de recursos, entonces sólo lo dejamos pasar hacia el *OpenCL* si hay recursos disponibles. En otro caso, se quedará bloqueado esperando recursos tras liberar el semáforo del gestor de recursos.

c) Se ejecutará la operación sobre el *OpenCL*.

d) Al terminar la operación de *OpenCL*, entramos al gestor de recursos bloqueando su semáforo.

e) Si la operación se tiene que repetir por falta de recursos, desbloqueamos el semáforo de *OpenCL* y nos ponemos a la espera, no sin antes desbloquear el gestor de recursos. Si la operación finalizó correctamente y ésta ha liberado recursos, entonces se desbloquea alguna de las operaciones que estén esperando recursos, si es el caso.

f) Finalmente, se sale del gestor de recursos, desbloqueando los semáforos de gestión de recursos y el de entrada al *OpenCL*.

El gestor de recursos prioriza las operaciones de liberación de recursos sobre las de solicitud, minimizando los bloqueos por falta de recursos disponibles. Para bloquear una operación por falta de recursos se han utilizado tres colas de espera, una para tareas asignadas a la *CPU*, una segunda para el resto de dispositivos y una última para tareas asignadas a todos los tipos de dispositivos.

6.3.5. Optimización de los recursos de OpenCL

En esta sección explicamos algunas alternativas que se han tomado para maximizar el uso de los recursos de *OpenCL*.

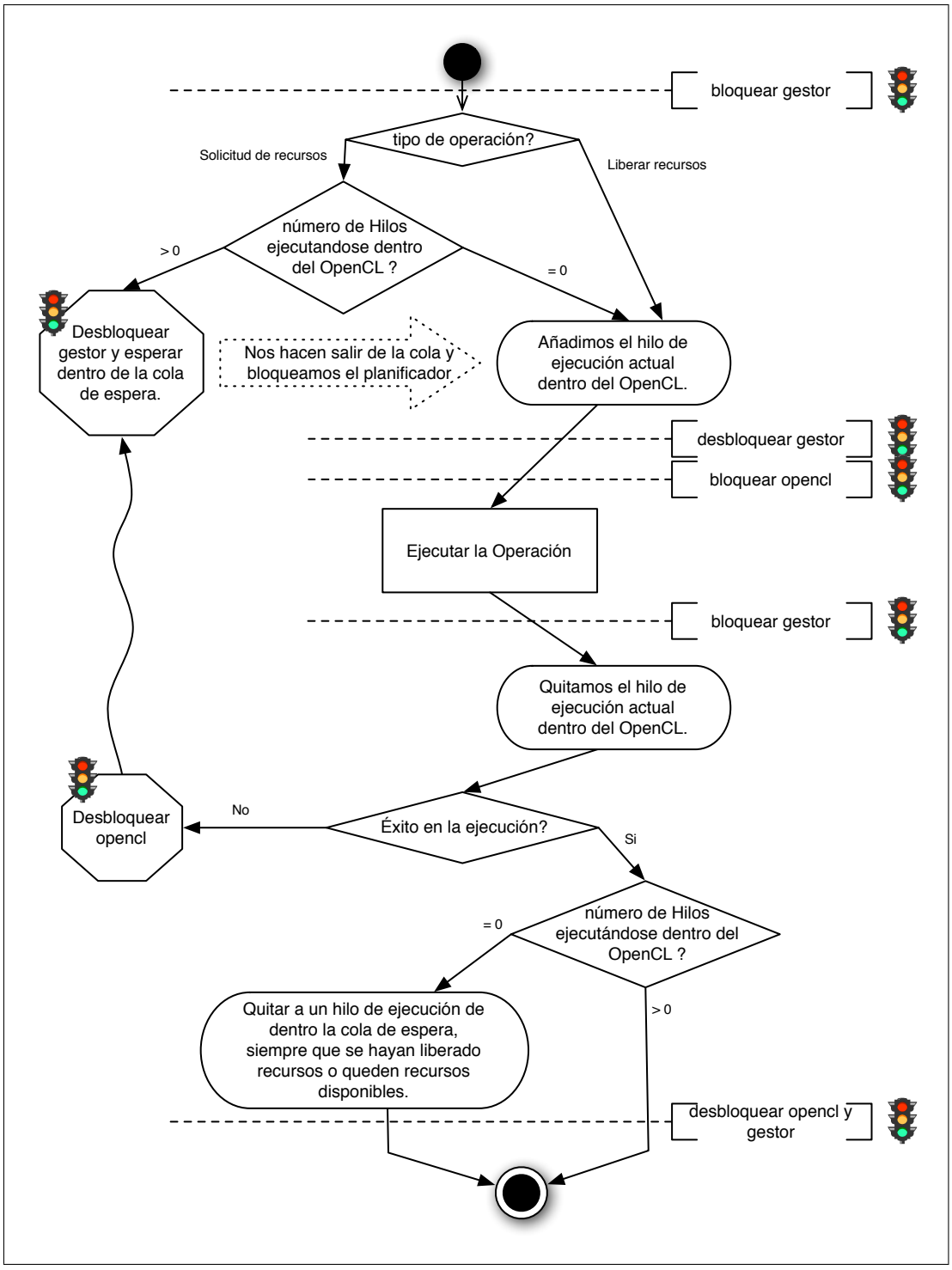

Figura 6.5: Planificador de las operaciones sobre *OpenCL*

Tipos de objetos de memoria y dispositivos

Para ejecutar un código dentro de un dispositivo se requiere inicializar una serie de objetos de memoria de *OpenCL*. Cada objeto de memoria se corresponde a un argumento de tipo puntero de la función que queremos ejecutar.

La *API* de *OpenCL* permite especificar si el objeto de memoria es de entrada, salida o de ambas cosas. Por otra parte se tiene que detallar también la política de memoria para el objeto. Hay dos posibles políticas:

1. Guardar los datos directamente dentro de la memoria del dispositivo, y
2. Utilizar directamente los datos del la memoria principal del sistema, con la posibilidad de que la implementación del *runtime* de *OpenCL* pueda aplicar técnicas de caché para acelerar su ejecución.

En la figura 6.6 podemos ver la selección del tipo de memoria según el tipo de dispositivo que se ha elegido en este proyecto. Si se trata de un dispositivo que trabaja con memoria propia[4], entonces escogemos la primera política. Con esta opción debemos solicitar explícitamente operaciones de lectura y escritura de datos.

En caso de que el dispositivo sea la *CPU*, escogemos la segunda política, ya que sabemos que este dispositivo utiliza directamente los datos almacenados en memoria principal. Con esta política nos ahorramos las transferencias de memoria. En cualquier caso, para ser consistentes con la *API* de *OpenCL*, será necesario sincronizar los datos una vez terminada la tarea.

Fisión de los dispositivos

En algunas pruebas que hemos realizado sobre las implementaciones actuales de *OpenCL* hemos visto limitaciones para ejecutar varias tareas a la vez. Afortunadamente, *OpenCL* nos ofrece un método para dividir un dispositivo en dispositivos más

[4]Nos referimos a la memoria del dispositivo que no es parte de la memoria principal del sistema

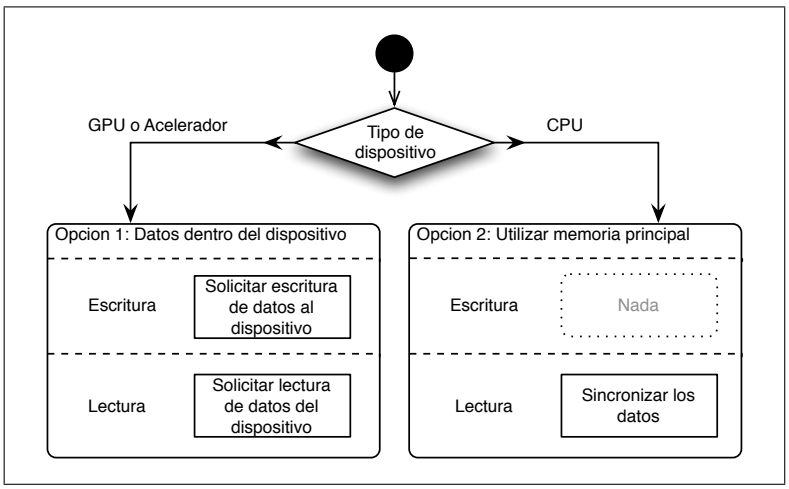

Figura 6.6: Elección del tipo de objetos de memoria de *OpenCL* según el tipo de dispositivo

pequeños llamados subdispositivos. De esta forma se logra un mayor control sobre los recursos disponibles, permitiéndonos ejecutar la tarea en más de un subdispositivo a la vez.

Para poder realizar una fisión de un dispositivo, primero será necesario que haya soporte para esa opción, ya que este método pertenece a una extensión de *OpenCL*[4].

Cuando hayamos dividido el dispositivo, podremos trabajar con los subdispositivos como si fuesen independientes. Para optimizar su uso, en nuestra implementación se crea un *context* para todos los subdispositivos de un dispositivo. Esto permite que al añadir un programa solo se tenga que compilar una vez para todos los subdispositivos.

Capítulo 7

Resultados

En este capítulo veremos primero el entorno de pruebas que hemos usado para comprobar la funcionalidad de nuestro sistema. Luego examinaremos las funcionalidades que nos ofrecen las implementaciones actuales de *OpenCL*, utilizando test sintéticos. Estos test de funcionalidad se desarrollan a lo largo del proyecto para tomar las mejores decisiones de diseño e implementación. Finalmente, haremos pruebas sobre dos aplicaciones reales: *Black-Scholes* y *Perlin Noise*.

7.1. Entorno de pruebas

El proyecto se ha desarrollado en dos entornos de ejecución. El primero de ellos es un ordendaor portátil, con las características que se muestran en la tabla 7.1. Este ordenador se utilizó en la primera etapa del proyecto.

Sistema operativo	Ubuntu 10.04 LTS - Lucid Lynx 32bits, kernel 2.6.32-29-generic-pae
Compilador binario	gcc 4.4.3
Controlador de las GPUs	NVidia 260.19.26
CPU	Intel(R) Core(TM) i5 CPU M 540 @ 2.53 GHz
GPU	NVIDIA GeForce GT 330M con 512 MB de GDDR3
Memoria RAM	8 GB DDR3 @ 1066 MHz
Disco Duro	500 GB a 5.400 RPM

Tabla 7.1: Características técnicas MacBook Pro 17' (mid-2010)

A medida avanzó el proyecto, apareció la necesidad de probarlo dentro de un entorno con varios dispositivos, con soporte de *OpenCL*, diferentes de la *CPU*. La solución más económica era montar un ordenador con varias *GPU*s. La principal dificultad fue encontrar una placa base con soporte para 2 zócalos del tipo *PCI Express x16* aprovechando el máximo las piezas ya disponibles (tipo de memoria y *CPU*). La tabla 7.2 muestra las características del entorno que montamos con varias *GPU*s asociadas.

Sistema operativo	Ubuntu 11.04 - Natty Narwhal 32bits, kernel 2.6.38-8-generic-pae
Compilador binario	gcc 4.5.2
Controlador de las GPUs	NVidia 270.41.06
CPU	Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83GHz
GPU(1)	NVIDIA GeForce GTX 470 con 1280 MB de GDDR5
GPU(2)	NVIDIA GeForce 9800 GT con 512 MB de GDDR3
Memoria RAM	4 GB DDR2 @ 1066 MHz
Disco Duro	500 GB a 7.200 RPM

Tabla 7.2: Características técnicas Ordenador Personal de sobremesa

Para realizar todos los experimentos se han realizado cinco ejecuciones, seleccionando el menor tiempo. Para calcular los tiempo se ha instrumentado el código con llamadas a las librerías del sistema.

7.2. Pruebas de ejecución concurrente

Las siguientes pruebas se han diseñado para ver las funcionalidades reales que nos ofrecen las implementaciones actuales del estándar *OpenCL*. Más concretamente, se pretende ver la capacidad que tiene *OpenCL* para ejecutar más de una tarea a la vez, ya sea sobre un o más dispositivos.

En las pruebas realizadas, las transferencias de datos son mínimas, ya que nos interesa fijarnos en el coste de ejecución de los *kernels*.

En las diferentes pruebas se han creado dos *kernels* sintéticos: uno para las *GPU*s y otro para la *CPU*s. Evitamos usar el mismo *kernel* para dispositivos distintos ya que la diferencia en tiempo de ejecución de un mismo *kernel* en dos dispositivos distintos es demasiado grande; haciendo difícil concluir algo de los resultados.

Además, la según la implementación de *OpenCL*, la forma de esperar la finalización de la ejecución puede variar el comportamiento final [10][3]. Debido a ello, en todos los test se usarán 3 formas distintas para esperar a la finalización de la ejecución: *a*) Usando la función *clWaitEvents* secuencialmente por cada evento disponible, *b*) usando la función *clWaitEvents* creando hilos de ejecución para esperar simultáneamente a todos los eventos creados, y *c*) usando la función *clFinish* creando hilos de ejecución para esperar a la finalización de las *commandQueues*.

7.2.1. Ejecución concurrente en varios dispositivos

En esta sección queremos ver el comportamiento de *OpenCL* cuando ejecuta *kernels* en todos los dispositivos disponibles simultáneamente. Para ello, se han inicializado todos los dispositivos y se les ha ordenado ejecutar un *kernel*.

En la figura 7.1 podemos ver los resultados del test. Primero hemos calculado el coste de ejecución del *kernel* para cada dispositivo por separado(las primeras filas de la tabla de la figura). Luego se han sumado todos los tiempos(cuarta fila de la

	Tiempo de ejecución (s)
Una ejecución - CPU (Q9550)	0,071330
Una ejecución - GPU [1] (GeForce GTX 470)	0,042102
Una ejecución - GPU (2) (GeForce 9800 GT)	0,377604
Tiempo acumulado	0,491036
Tiempo máximo	0,377604
Test - clWaitEvents secuencial	0,373862
Test - clWaitEvents con hilos de ejecución	0,373862
Test - clFinish con hilos de ejecución	0,371147

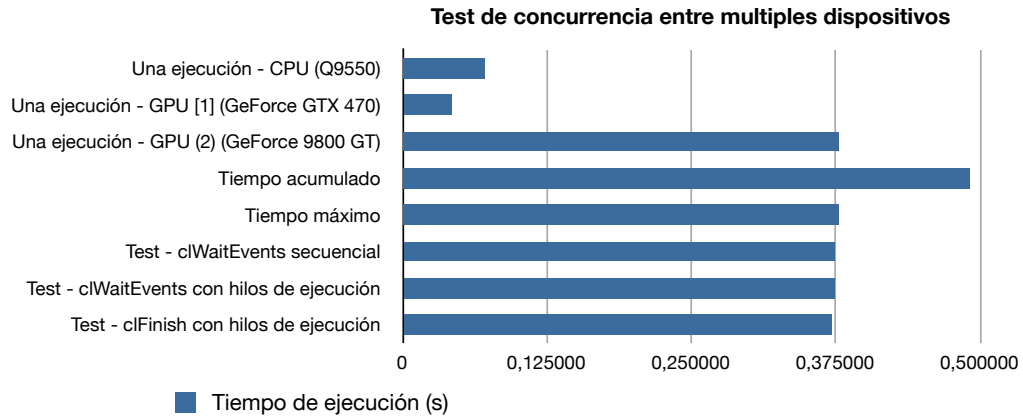

Figura 7.1: Resultados del test de concurrencia entre múltiples dispositivos en el PC personal

figura 7.1), para tener una referencia del tiempo necesario para ejecutar los *kernels* secuencialmente, uno detrás del otro. La quinta fila de la tabla(Tiempo Máximo) es el tiempo del dispositivo que tarda más. Este debería ser el tiempo de ejecución de todos los *kernels* si estos se pueden ejecutar en paralelo.

Finalmente, las últimas filas de la tabla muestran el tiempo de ejecución de los *kernles* en los tres dispositivos a la vez, utilizando las tres modalidades de espera diferentes.

Los resultados confirman que la implementación de *OpenCL* está ejecutando concurrentemente los *kernels* entre múltiples dispositivos, ya que el tiempo de los test es prácticamente igual al dispositivo que más tiempo emplea para ejecutar el *kernel*.

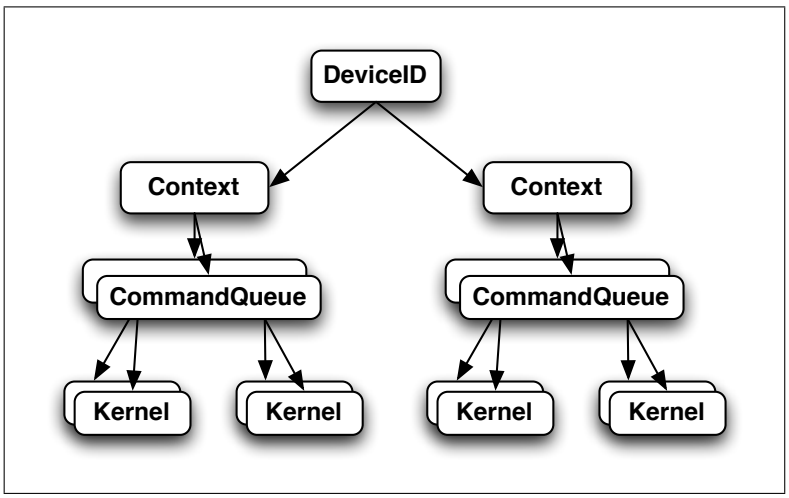

Figura 7.2: Relación entre los componentes de *OpenCL* para hacer el test de concurrencia en un dispositivo

7.2.2. Ejecución concurrente en un dispositivo

El objetivo de este test es poner a prueba *OpenCL* para ver si es capaz de ejecutar múltiples *kernels* concurrentemente dentro de un mismo dispositivo.

Para llevar a cabo el test, se ha planteado el caso más simple para que *OpenCL* intente ejecutar un *kernel* concurrentemente. Consiste en ejecutar el mismo *kernel* utilizando la misma o diferentes *commandQueue*, y el mismo o diferentes *context*. En la figura 7.2 vemos cómo se relacionan los componentes de *OpenCL* para hacer el test. Hemos declarado dos *context* con dos *commandQueues* para cada *context* y hemos ejecutando dos *kernels* dentro de cada *commandQueue*. Así, en total, se ejecutarán un total de ocho *kernels*.

En las figuras 7.3, 7.4 y 7.5 vemos los resultados del test para los dispositivos *CPU*, *GPU(1)* y *GPU(2)* correspondientes al PC personal detallado en la tabla 7.2, respectivamente.

En cada una de ellas se muestra el tiempo de ejecución de un único *kernel* en el dispositivo(primera fila de la tabla), el tiempo estimado en ejecutarse los 8 *kernels* uno detrás del otro sin explotar concurrencia (segunda fila) y, los tiempos reales de la

CPU (Q9550)	Tiempo de ejecución (s)
Una ejecución	0,359845
Tiempo Estimado sin concurrencia	2,878760
Test - clWaitEvents secuencial	2,844190
Test - clWaitEvents con hilos de ejecución	2,845640
Test - clFinish con hilos de ejecución	2,844360

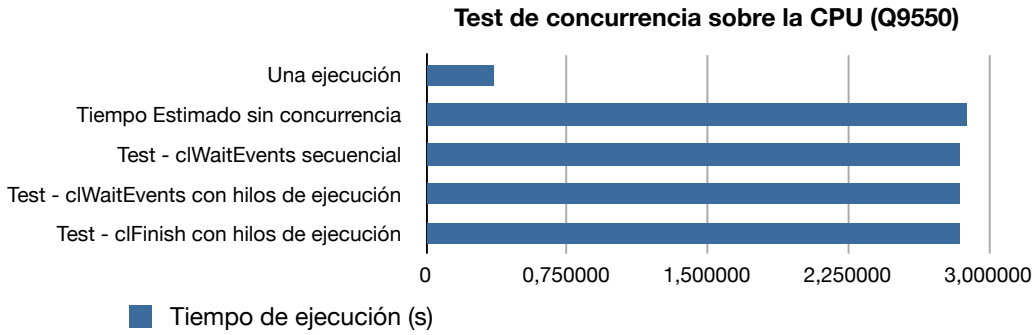

Figura 7.3: Resultados del test de concurrencia en la *CPU* del PC personal

ejecución de los 8 *kernels* en el dispositivo utilizando las tres modalidades distintas de esperarnos (tres últimas filas de las tablas).

Como podemos ver en los resultados, cada dispositivo ha consumido el mismo tiempo tanto para ejecutar los tests (tres últimas filas), como para ejecutar secuencialmente los ocho *kernels*(segunda fila). Esto pone de manifiesto las limitaciones de la implementación de *OpenCL* para la *CPU*. La *CPU* que estamos usando para realizar el test tiene cuatro núcleos y evidentemente permite la ejecución concurrente. En el caso de las *GPU*s es un tema distinto, ya que éstas, además, tienen restricciones a nivel *hardware*.

7.2.3. Ejecución concurrente creando subdispositivos

En la sección 6.3.5 planteamos dividir en subdispositivos un dispositivo de *OpenCL* para explotar sus recursos. El objetivo del test es comprobar que efectivamente con este sistema podemos aprovechar mejor los recursos disponibles.

GPU [1] (GeForce GTX 470)	Tiempo de ejecución (s)
Una ejecución	0,042113
Tiempo Estimado sin concurrencia	0,336904
Test - clWaitEvents secuencial	0,335471
Test - clWaitEvents con hilos de ejecución	0,335516
Test - clFinish con hilos de ejecución	0,335559

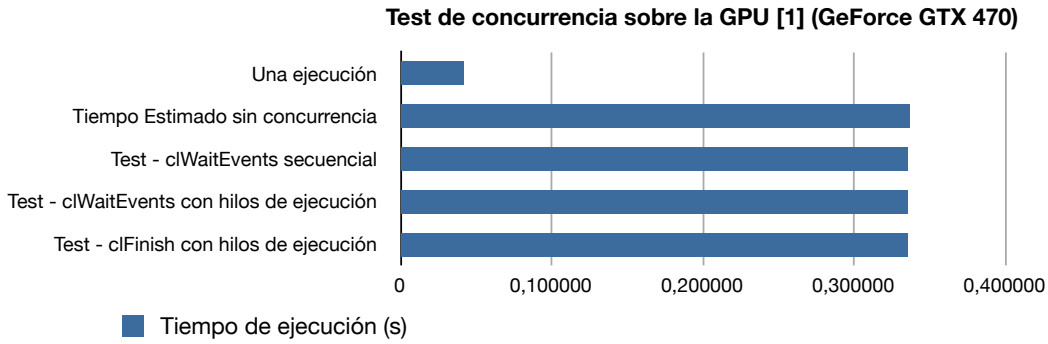

Figura 7.4: Resultados del test de concurrencia en la primera *GPU* del PC personal

GPU [2] (GeForce 9800 GT)	Tiempo de ejecución (s)
Una ejecución	0,377796
Tiempo Estimado sin concurrencia	3,022368
Test - clWaitEvents secuencial	3,001090
Test - clWaitEvents con hilos de ejecución	3,001390
Test - clFinish con hilos de ejecución	3,017270

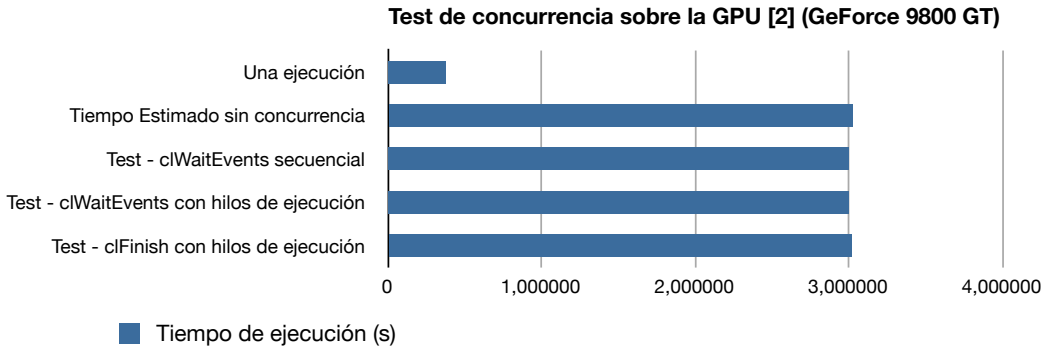

Figura 7.5: Resultados del test de concurrencia en la segunda *GPU* del PC personal

El test sólo se ejecutará en dispositivos con soporte de la extensión de fisión de *OpenCL*[4].

En la figura 7.6 vemos el resultado del test de concurrencia sobre los subdispositivos de la *CPU* del PC personal descrito en la tabla 7.2. Como vemos en la figura, se han creado cuatro subdispositivos, uno para cada núcleo de la *CPU* (primeras cuatro filas de la tabla de la figura 7.6). La quinta fila muestra el tiempo estimado en caso de que los *kernles* no se puedan ejecutar concurrentemente. Además, hemos visto interesante la posibilidad de probar el test compartiendo regiones de memoria para todos los *kernels* y para las tres modalidades de espera (filas seis a la ocho de la tabla). Las filas nueve a la once muestran los resultados en caso de tener regiones diferentes de memoria para los *kernels*.

En el caso de utilizar regiones de memoria diferentes, si se utilizan hilos de ejecución para llamar simultáneamente a *clWaitEvents* o *clFinish*(filas diez y once) se obtienen tiempos de ejecución muy parecidos al de ejecutar un solo *kernel*. En estos casos podemos confirmar que los *kernels* se están ejecutando concurrentemente. En cambio, el tiempo de ejecución del caso en que esperamos a los eventos de forma secuencial (fila ocho de la tabla) tarda lo mismo que la ejecución secuencial de los *kernels*. Esto es debido a que no se empiezan a ejecutar los *kernels* hasta que no se espera explicitamente la finalización de los eventos. Como ya habíamos comentado en la sección 7.2, las implementación del *OpenCL* son sensibles a la forma de esperar la finalización de los eventos.

Cuando se utiliza la misma región de memoria (filas seis a la ocho de la tabla de la figura 7.6) el tiempo de ejecución es superior a la ejecución secuencial de los *kernels*. Esto nos indica que cada núcleo de la *CPU* esta luchando para acceder a las mismas direcciones de memoria, activando sincronismos que tiene un coste adicional con tal de mantener la consistencia de los datos.

CPU (Q9550)	Tiempo de ejecución (s)
Una ejecución - CPU[1] (subdispositivo 1)	0,073081
Una ejecución - CPU[2] (subdispositivo 2)	0,071229
Una ejecución - CPU[3] (subdispositivo 3)	0,071229
Una ejecución - CPU[4] (subdispositivo 4)	0,071232
Tiempo Estimado sin concurrencia	0,286771
Test - clWaitEvents secuencial - misma región de memoria	0,284670
Test - clWaitEvents con hilos de ejecución - misma región de memoria	0,491435
Test - clFinish con hilos de ejecución - misma región de memoria	0,447220
Test - clWaitEvents secuencial - diferentes regiones de memoria	0,284789
Test - clWaitEvents con hilos de ejecución - diferentes regiones de memoria	0,095870
Test - clFinish con hilos de ejecución - diferentes regiones de memoria	0,080986

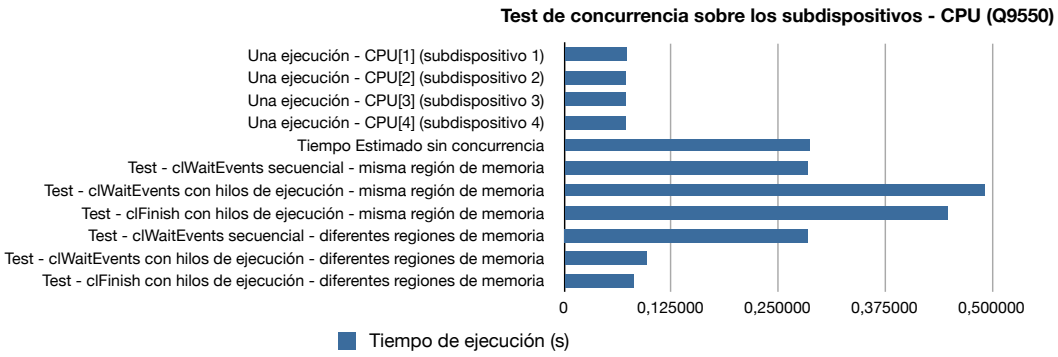

Figura 7.6: Resultados del test de concurrencia sobre los subdispositivos de la *CPU* en el PC personal

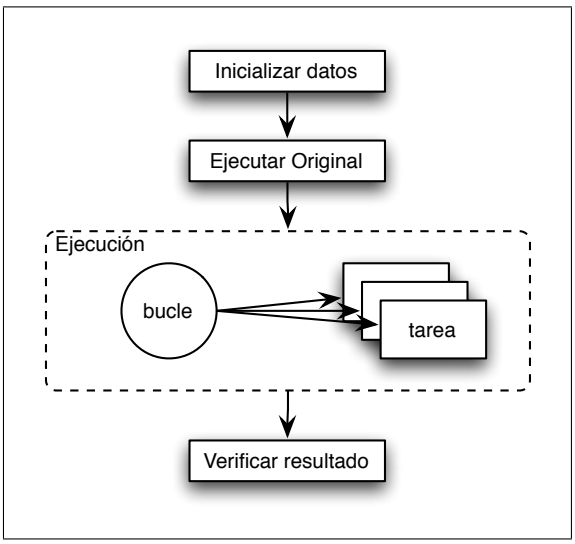

Figura 7.7: Esquema de la ejecución de las aplicaciones

7.3. Pruebas del proyecto

Figura 7.7 muestra el esquema que se ha seguido para las aplicaciones que hemos probado. Primero de todo se inicializan los datos de entrada y se ejecuta el código original para una comprobación posterior de los resultados. Después ejecutamos repetidamente una función de la aplicación con *pragma*s (es una función que constituye un tanto por ciento muy significativo del tiempo de ejecución del programa) para utilizar nuestro *runtime*. Finalmente, comprobamos los resultados de salida.

7.3.1. Perlin Noise

Perlin Noise es una aplicación que ejecuta una función matemática que genera valores pseudo-aleatoriamente a partir de unos datos de entrada. Con ello se pretende simular el ruido blanco o variabilidad en todo tipo de fenómenos[13].

El código 7.1 muestra la función de la aplicación *Perlin* que hemos anotado con las construcciones de *OmpSs*. Nuestro sistema interpreta estas construcciones para poder usar *OpenCL*.

```
1 #pragma omp target device(cpu,gpu) copy_deps
2 #pragma omp task output( [elementsPerKernel] output )
3 void compute_perlin_noise_continous_region(pixel * output, const
      float time, int x, int y, int region_width)
4 { ... }
```

Código 7.1: Declaración de los pragmas para la función del Perlin

La figura 7.8 muestra los resultados de tiempo de ejecución en función del número de hilos usados en la ejecución. En la figura se muestra los tiempos tanto para la aplicación usando nuestro sistema de compilación (ejecución con *OmpSs*$_{CL}$) como la aplicación sin usar nuestro sistema (ejecución sin *OmpSs*$_{CL}$).

Primero nos fijaremos en la tabla y en los gráficos de sectores de la figura. Para el caso de la ejecución con *OmpSs*$_{CL}$, los resultados muestran que se usan únicamente las *GPU*s para la ejecución de los *kernels* (0 % para las *CPU*s) cuando se usan poco hilos en la ejecución (1 o 2 hilos). Esto es debido a que el planificador prioriza el uso de *GPU*s sobre las *CPU*s. Cuando se tienen más de dos hilos de ejecución, se empiezan a usar las *CPU*s, bajando el tiempo total de ejecución significativamente. Por otro lado, a medida que aumentamos el número de hilos de ejecución, se incrementa el tiempo de ejecución del código original sin utilizar construcciones *OmpSs*$_{CL}$. Esto es a causa de la arquitectura del procesador y del *runtime* de *Nanos*, que crea un conjunto de hilos de ejecución que absorben recursos del sistema, aunque estos no se usen para ejecutar ninguna tarea.

La figura 7.9 muestra las porciones de tiempo consumido para cada dispositivo para ejecutar el *kernel* del *Perlin*. Los tiempos que se muestran son los de escribir *buffers*[1], de ejecución de los *kernels*, y el de leer *buffers*[2]. Como podemos observar, se trata de una operación que precisa poca transferencia de memoria. En el caso de la *CPU*, ésta no tiene la necesidad de transferir datos ya que la memoria de los dispositivos

[1]Tiempo empleado para transferir los datos de entrada para la ejecución del *kernel*
[2]Tiempo empelado para transferir los resultados, una vez ejecutado el *kernel*, a la memoria principal del sistema

Número de hilos	Tiempo de ejecución con OmpSs_cl (s)	Tiempo de ejecución sin aceleración (s)	CPU (subdividida)	GPU[1]	GPU[2]
1	4,57	0,47	0%	100%	0%
2	2,85	0,48	0%	61%	39%
4	0,43	0,47	87%	8%	5%
8	0,45	2,41	93%	4%	2%
16	0,55	8,47	90%	6%	4%

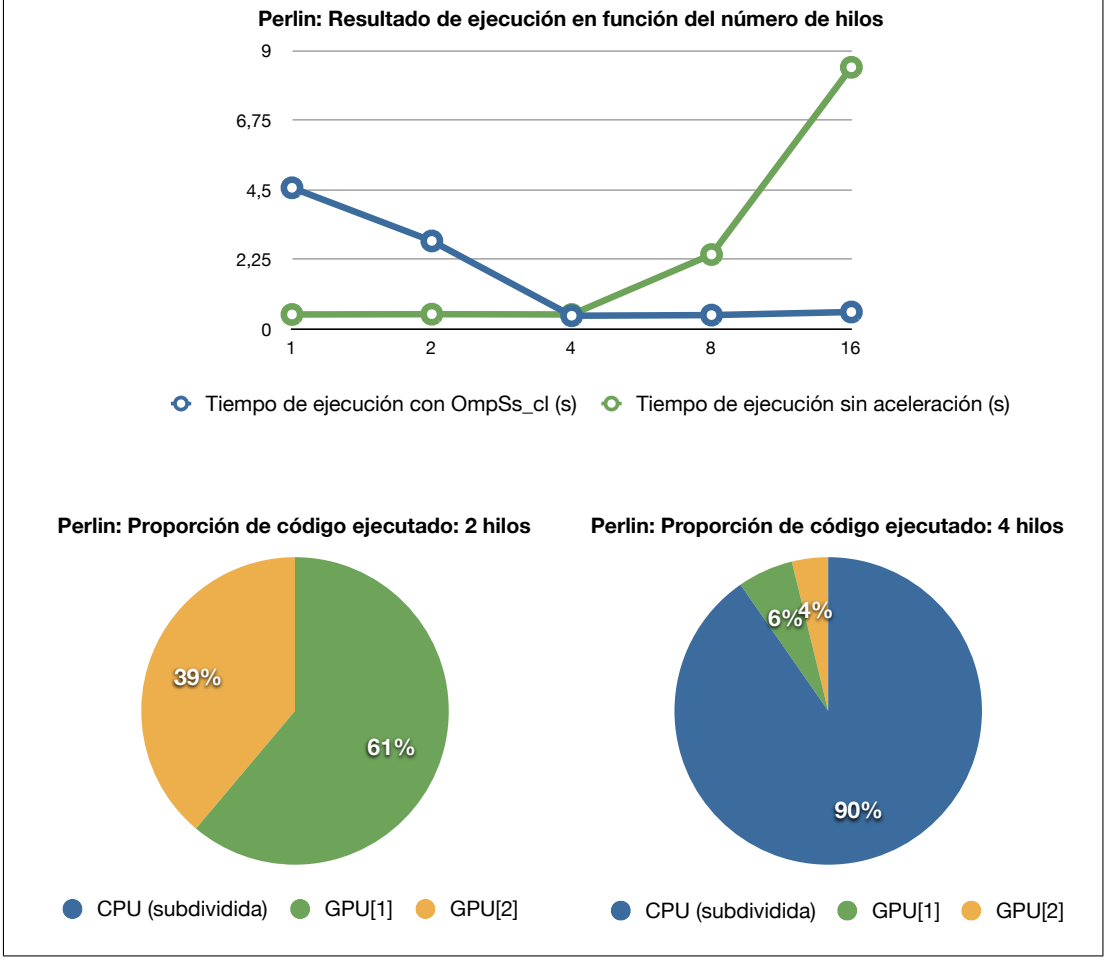

Figura 7.8: Resultados del algoritmo *Perlin*, variando el número de hilos de ejecución.

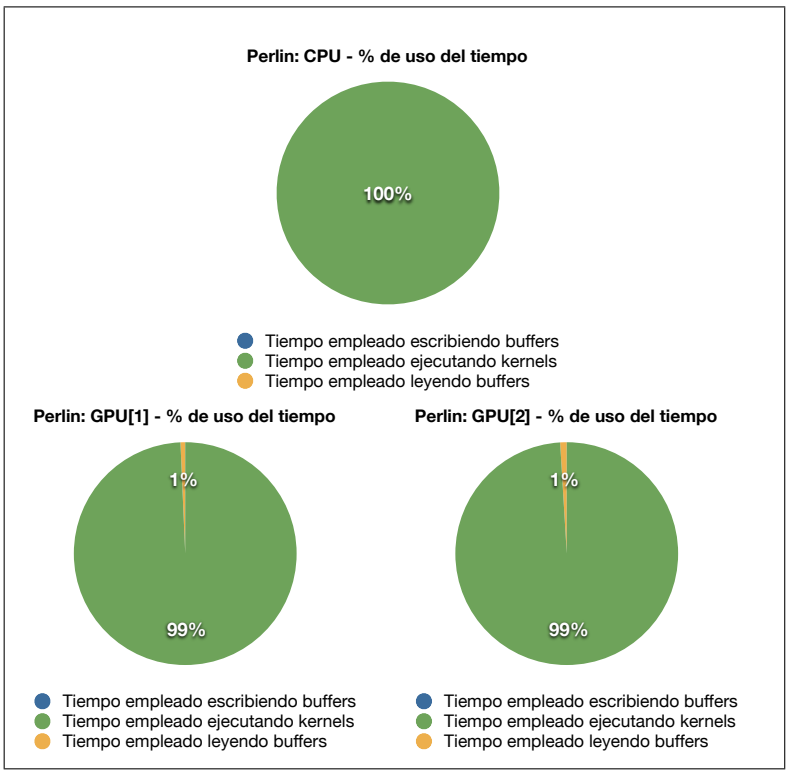

Figura 7.9: Perlin: Porcentajes de tiempo consumido para la ejecución del *kernel*, incluyendo las transferencias de datos

coincide con la memoria del procesador *host*.

Figura 7.10 muestra los tiempos de ejecución de los *kernels* cuando usan o no operaciones *SIMD*. Las versiones vectorizadas aparecen en la figura como "vectorial" y, en caso de no usar esas operaciones, aparecen como "escalar". En los resultados podemos ver una mejora en el tiempo de ejecución de la *CPU*. Sin embargo, para las *GPU*s que tenemos en nuestro entorno de trabajo, el código vectorizado para *OpenCL* es sensiblemente más lento. Esto está documentado en el apéndice A, que indica que las *GPU*s prefieren operaciones escalares.

A continuación, para analizar en detalle el uso que se hace de nuestro *runtime* en la ejecución de los *kernel*s, mostraremos algunos resultados de nuestra instrumentación. En la figura 7.11 vemos la leyenda de los colores de las figuras que veremos a continuación. Nótese que la leyenda se ha divido en dos partes. La parte superior de la leyenda describe la parte superior de las figuras 7.12-7.14 donde se muestra el

Experimento	Tiempo de ejecución (s)	CPU (subdividida)	GPU[1]	GPU[2]
Sin aceleración	0,47	--	--	--
(CPU,GPU) + escalar	0,43	87%	8%	5%
(CPU,GPU) + vectorial	0,30	95%	4%	1%
(GPU) + escalar	3,45	--	61%	39%
(GPU) + vectorial	9,16	--	86%	14%
(CPU) + escalar	0,26	100%	--	--
(CPU) + vectorial	0,16	100%	--	--

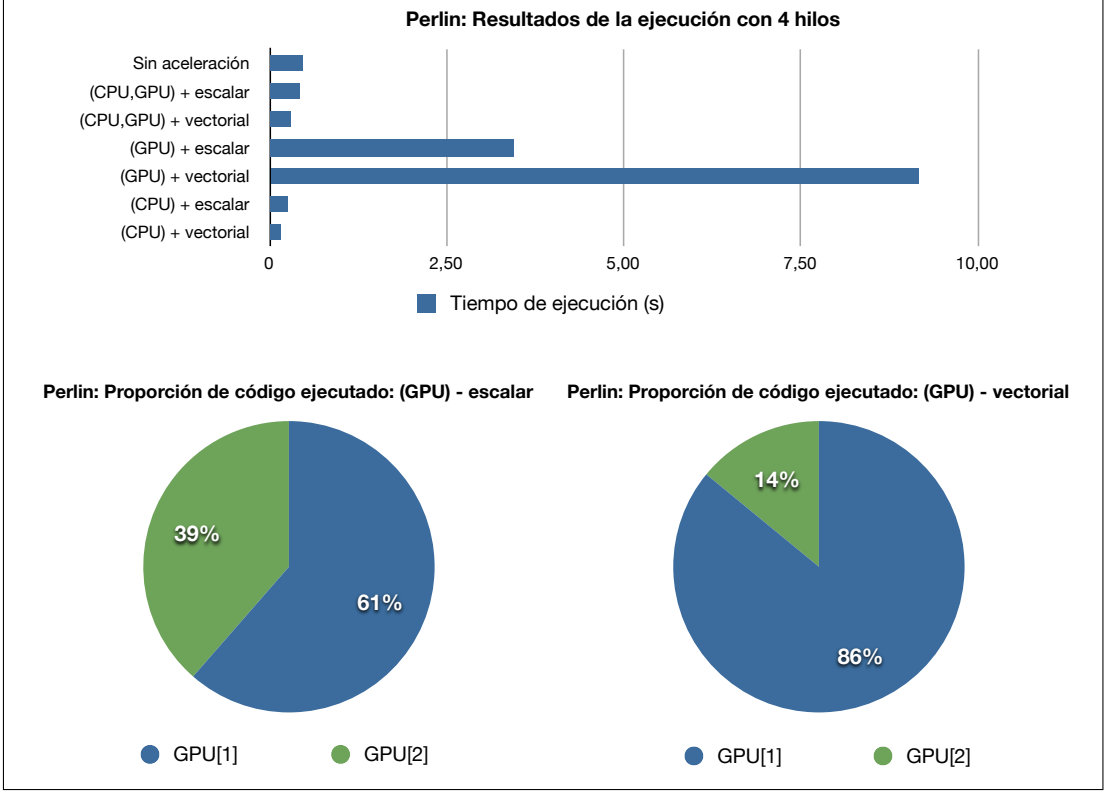

Figura 7.10: Resultados del algoritmo *Perlin*, usando 4 hilos de ejecución, variando los dispositivos y *kernels*

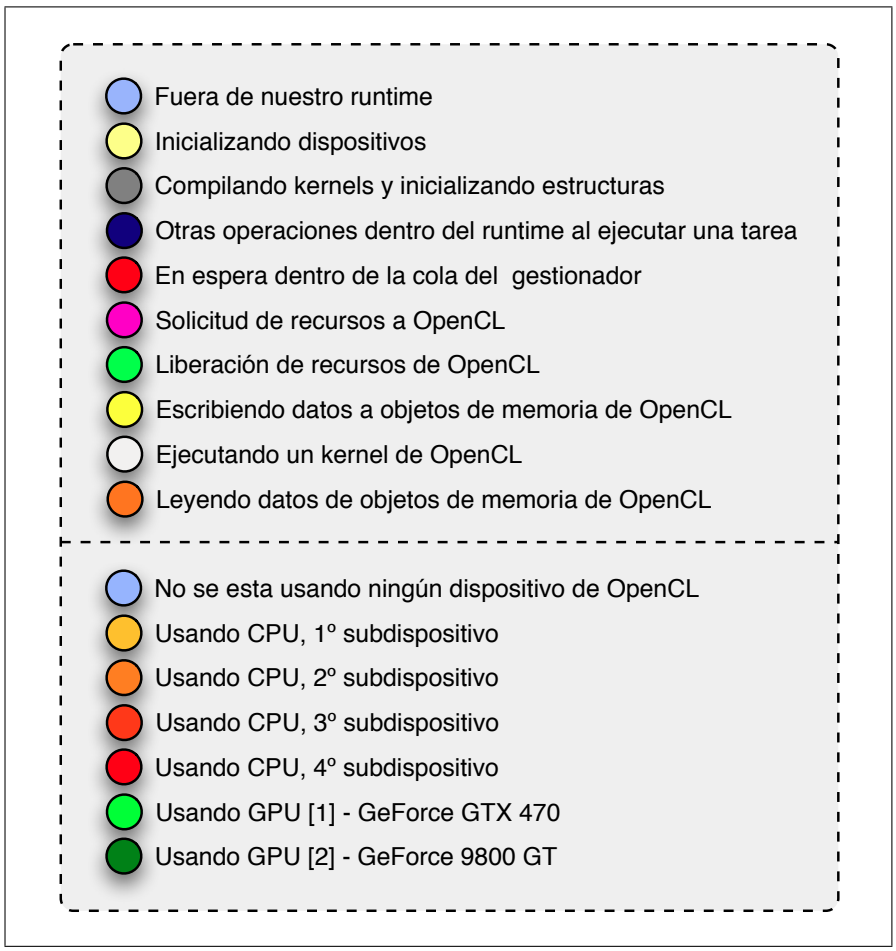

Figura 7.11: Leyenda para las figuras 7.12, 7.13 y 7.14

estado de nuestro *runtime*. La parte inferior de la leyenda describe la parte inferior de las figuras 7.12-7.14, donde se muestran los dispositivos de *OpenCL* que se están usando.

La figura 7.12 muestra la ejecución completa de la aplicación *Perlin*, con los diferentes hilos utilizados en la ejecución de la aplicación a lo largo del tiempo. Podemos distinguir cinco partes señaladas en la figura:

1. Nuestro *runtime* inicializa los dispositivos de *OpenCL* disponibles en el sistema.
2. Se inicializan estructuras de datos y se compilan los *kernels*.
3. Se inicializan los datos de entrada de la aplicación y se ejecuta el algoritmo original sin *pragma*s.

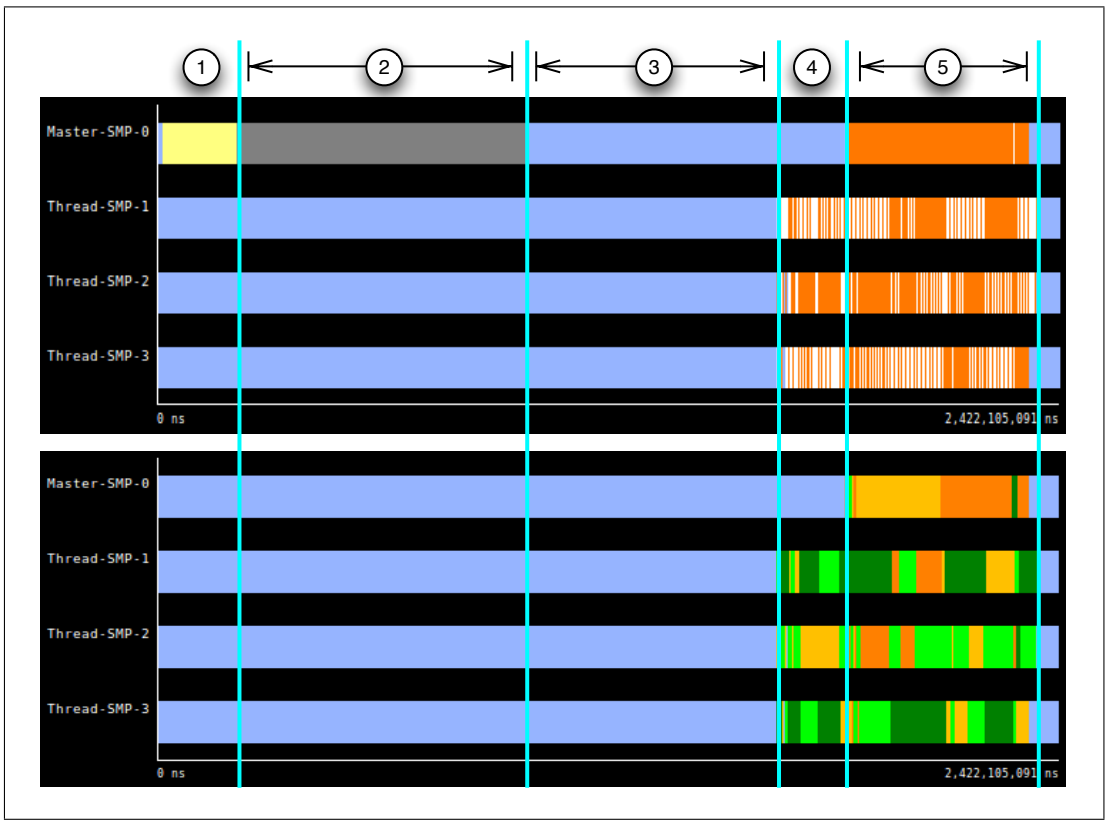

Figura 7.12: Traza *paraver* de la ejecución del *Perlin*

4. El hilo principal de ejecución encola tareas dentro del *runtime* de *Nanos* y los otros hilos empiezan a usar nuestro *runtime* para realizar operaciones sobre *OpenCL*.

5. Todos los hilos de ejecución se están usando para ejecutar tareas, hasta finalizar los cálculos.

.

La figura 7.13 muestra un fragmento de la quinta parte de la figura 7.12. Observando la parte inferior de la figura 7.13, podemos ver que la *CPU* está ejecutando los *kernels* más deprisa y más veces que las *GPU*s. Esto nos hace pensar que el uso que se está haciendo de la *GPU*s por parte de los kernels implementados no es el más eficiente posible.

Para analizar en detalle el funcionamiento de nuestro *runtime* hemos realizado una

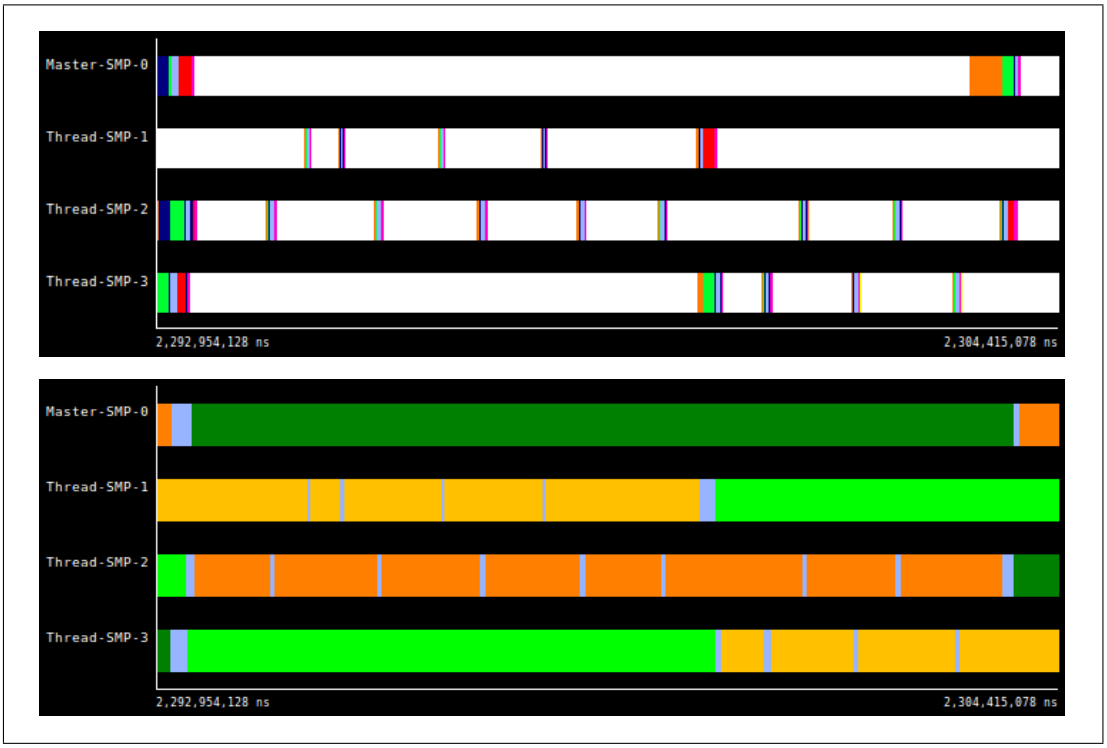

Figura 7.13: Quinto fragmento de la traza *paraver* de la figura 7.12

aproximación a una parte de la ejecución mostrada en la figura 7.13. La figura 7.14 muestra la ejecución de la aplicación en esta aproximación. Así, en el punto 1 de la figura, podemos ver parte del funcionamiento del gestor de recursos de *OpenCL* explicado en el sección 6.3.4. En este punto, el segundo hilo de ejecución produce una operación de solicitud de recursos, y seguidamente, el hilo 4 también los solicita. Como se puede observar, el hilo 4 se bloquea en la cola de espera hasta que se llega al punto 2 de la figura. En ese punto, el hilo 2 libera los recursos que podrá usar el hilo 4.

7.3.2. Black-Scholes Options Pricing

Black-Scholes es un algoritmo empleado para estimar el valor actual de una opción europea para la compra o venta de acciones en una fecha futura[12].

A diferencia del *Perlin*, *Black-Scholes* tiene datos de entrada no escalares, es decir que se deberán escribir vectores de datos en los *buffers* para poder ejecutar el kernel

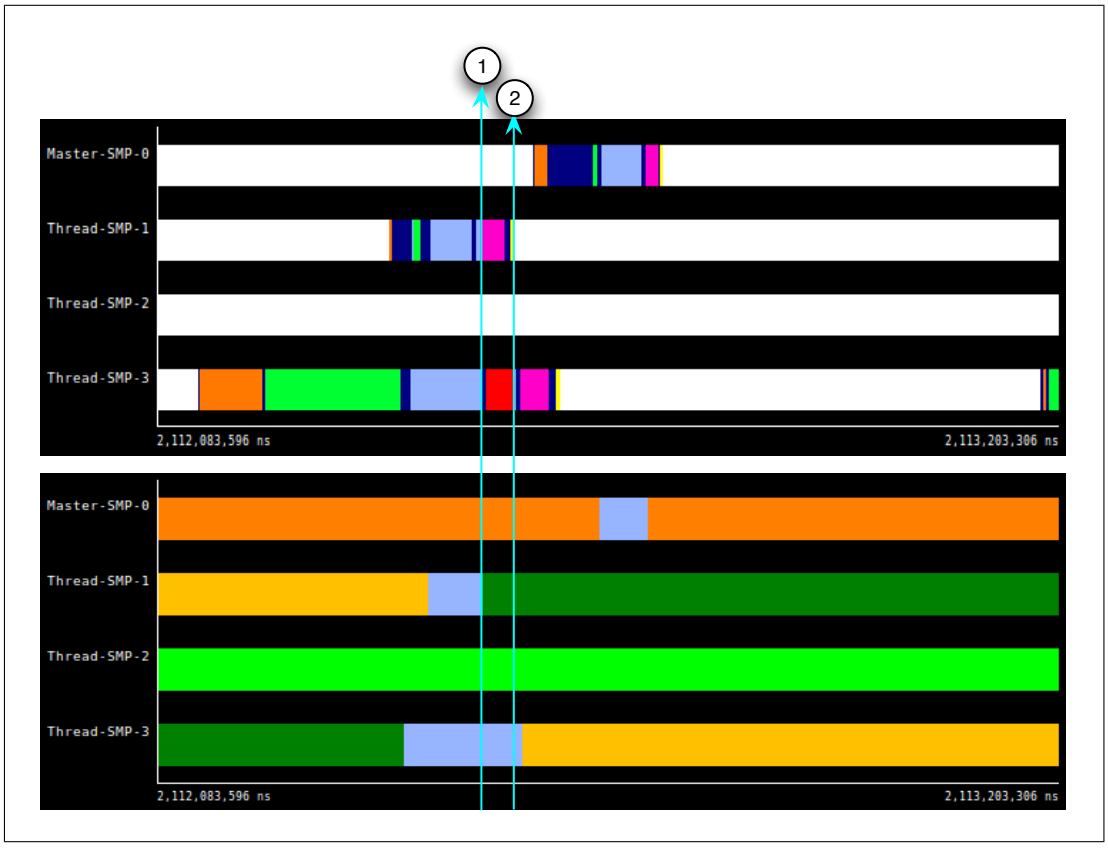

Figura 7.14: Aproximación de la traza *paraver* de la figura 7.13

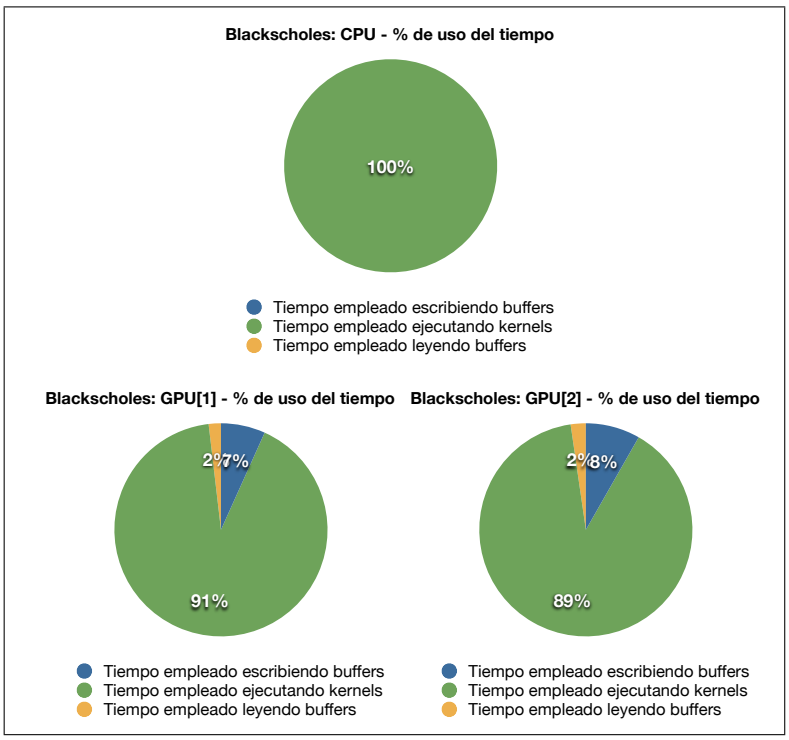

Figura 7.15: Black-Scholes: Porcentajes de tiempo consumido para la ejecución del *kernel*, incluyendo las transferencias de datos

en el dispositivo. En la figura 7.15 vemos que el consumo de tiempo para escribir los *buffers* es mayor que en el caso del *Perlin*.

Figura 7.16 y 7.17 muestran los resultados muy similares a los resultados que mostramos para la aplicación *Perlin* con las figuras 7.8 y 7.10: las *GPU*s se usan cuando hay pocos hilos de ejecución, se obtienen mejor rendimiento cuando se usan las *CPU*, la vectorización sólo ayuda a mejorar la ejecución en la *CPU*, etc.

A la vista de que los resultados son similares no se ha creído necesario realizar un análisis más en detalle con el *Paraver*.

Número de hilos	Tiempo de ejecución con OmpSs_cl (s)	Tiempo de ejecución sin aceleración (s)	CPU (subdividida)	GPU[1]	GPU[2]
1	7,25	1,16	0%	100%	0%
2	4,64	1,17	0%	63%	37%
4	0,88	1,19	85%	9%	6%
8	0,92	6,64	92%	5%	3%
16	0,96	21,08	93%	4%	3%

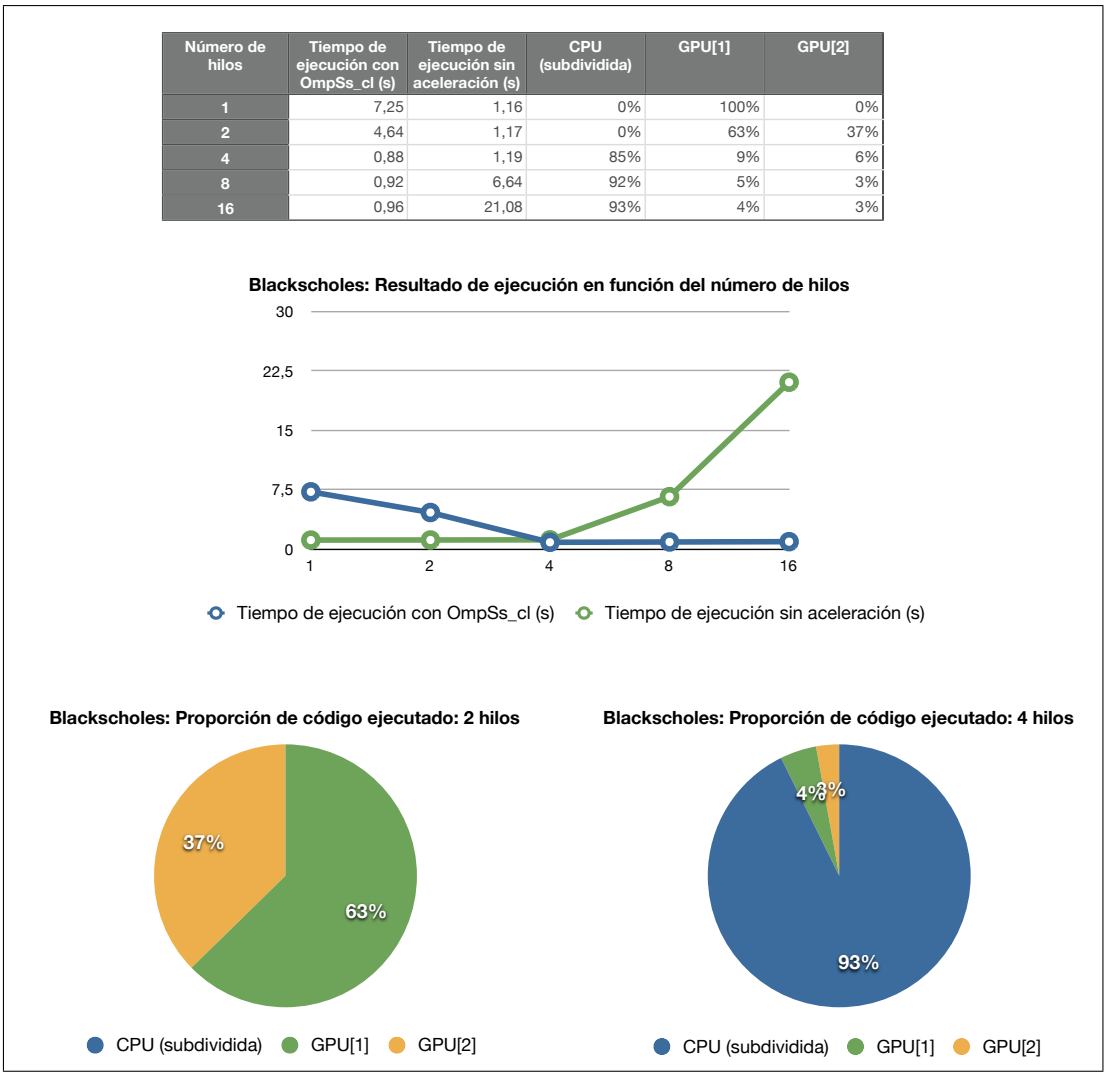

Figura 7.16: Resultados del algoritmo *Blackscholes*, variando el número de hilos de ejecución.

Experimento	Tiempo de ejecución (s)	CPU (subdividida)	GPU[1]	GPU[2]
Sin aceleración	1,16	--	--	--
(CPU,GPU) + escalar	0,88	85%	9%	6%
(CPU,GPU) + vectorial	0,81	85%	9%	6%
(GPU) + escalar	5,92	--	62%	38%
(GPU) + vectorial	5,64	--	62%	38%
(CPU) + escalar	0,56	100%	--	--
(CPU) + vectorial	0,52	100%	--	--

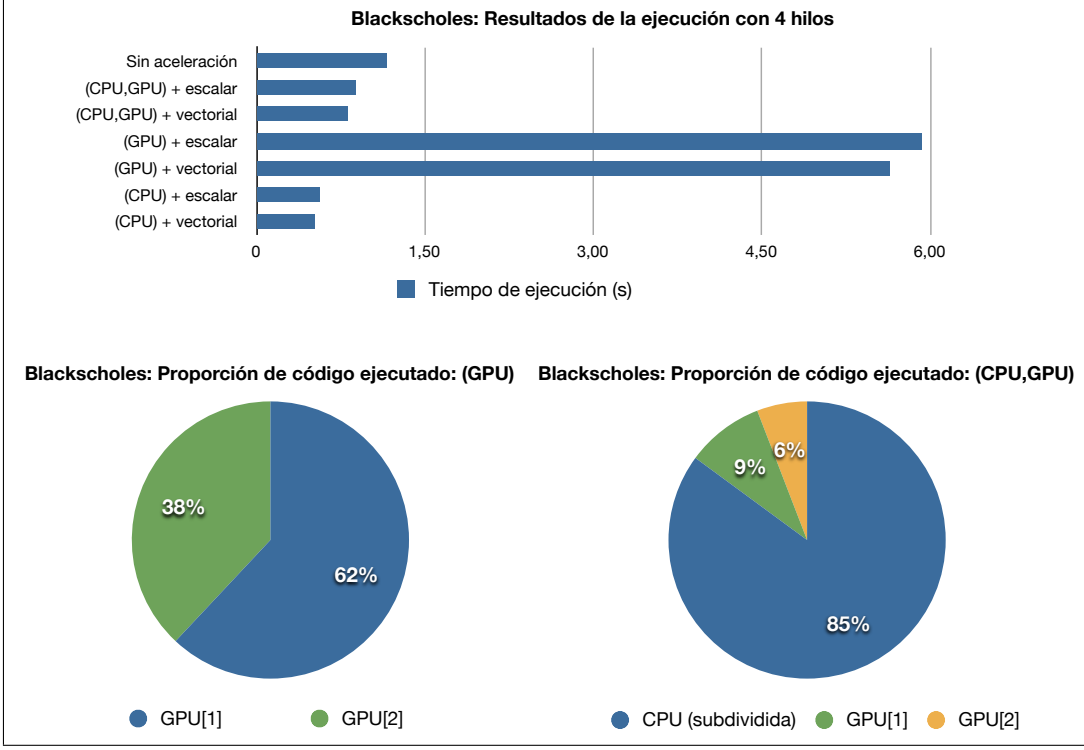

Figura 7.17: Resultados del algoritmo *Blackscholes*, usando 4 hilos de ejecución, variando los dispositivos y *kernels*

Capítulo 8

Planificación y coste

En esta sección realizaremos un cálculo del coste del proyecto, detallando las tareas que se han realizado con su planificación. Se han dividido los costes en tres categorias:

1. **Recursos humanos**: En la tabla 8.1 se detallan los costes del personal necesario para desarrollar el proyecto. Tenemos en cuenta 3 roles diferentes: *a*) Coordinador: encargado de supervisar el proyecto y delimitar las tareas para que se correspondan con la planificación. *b*) Analista: especifica y diseña el proyecto. *c*) Programador: realiza la implementación del proyecto.

Rol	Coste unitario	Dedicación	Coste total
Coordinador	60 € / hora	105	€ 6.300
Analista	40 € / hora	530	€ 21.200
Programador	35 € / hora	290	€ 10.150
	Total	925	€ 37.650

Tabla 8.1: Coste de los recursos humanos

2. **Recursos materiales**: En la tabla 8.2 se especifica el material amortizable que se ha usado.

Descripción	Precio	Amortización	Coste unitario	Coste total
PC personal	€ 1.600	4 años	33,34 € / mes	€ 150
Portátil Mac	€ 2.000	4 años	41,47 € / mes	€ 187
Otros	€ 2.500	10 años	20,83 € / mes	€ 94
			Total	€ 430

Tabla 8.2: Coste de los materiales amortizables

3. **Material fungilbe**: En la tabla 8.3 presentamos los costes del material fungible utilizado a lo largo de todo el proyecto.

Concepto	Coste
Material de oficina	€ 60
Internet / teléfono	€ 120
Total	€ 180

Tabla 8.3: Coste de los materiales fungibles

En la tabla 8.4 podemos ver la planificación final para realizar las tareas necesarias para completar el proyecto. El proyecto ha tenido una duración total de 4 meses y medio para terminar todas las tareas asignadas.

Tarea	Horas	Fecha inicio	Fecha fin
Especificación	80	20/02/11	16/03/11
Análisis	172	28/02/11	01/05/11
Diseño	148	12/03/11	07/05/11
Implementación	210	01/04/11	23/05/11
Pruebas	80	01/05/11	14/06/11
Documentación	195	15/03/11	22/06/11
Preparación de la defensa	40	23/06/11	29/06/11
Total	925		

Tabla 8.4: Planificación final

Finalmente en la tabla 8.5 presentamos el resumen del coste total del proyecto.

Concepto	Coste
Recursos humanos	€ 37.650
Recursos materiales	€ 430
Material fungible	€ 180
Total	€ 38.260

Tabla 8.5: Coste total del proyecto

Capítulo 9

Conclusiones

En este proyecto hemos desarrollado *OmpSs*$_{CL}$. *OmpSs*$_{CL}$ es una infraestructura de compilación y ejecución que facilita la programación de un sistema heterogéneo por medio de las construcciones de *OmpSs*. El *runtime* de *OmpSs*$_{CL}$ lo hemos implementado sobre un estándard reconocido, *OpenCL*, que soporta este tipo de sistemas heterogéneos. En el desarrollo de *OmpSs*$_{CL}$ se han integrado varios sistemas: *Nanos runtime*, *Mercurium*, *OpenCL* del *SDK* de *ATI*[1] y *OpenCL* del *SDK* de *NVidia*[2]. El resultado de este desarrollo es un paquete cuyo proceso de instalación se ha automatizado.

Durante el desarrollo del proyecto se han analizado las implementaciones actuales de *OpenCL* tanto con pruebas sintéticas como con aplicaciones reales. Para facilitar ese análisis se han desarrollado unos módulos para la instrumentación del código del sistema con trazas *paraver*, y se han añadido opciones al compilador que permiten obtener información para el análisis y depuración del sistema.

Los resultados de estos análisis muestran que el rendimiento obtenido depende considerablemente de las implementaciones actuales de *OpenCL* y del código fuente de los *kernels*. Dos posibles razones del bajo rendimiento de la ejecución de un *kernel*

[1] Soporta la especificación de OpenCL 1.1
[2] Soporta la especificación de OpenCL 1.0

en una *GPU* son: (1) que la vectorización realizada por *OpenCL* no favorece a las *GPU* usadas, y (2) la imposibilidad de ejecutar más de un *kernel*, de forma concurrente, en un dispositivo. Así, aunque el sistema es totalmente funcional, creemos que quedan abiertos retos de optimización del sistema como el desarrollo de nuevas construcciones con *pragma*s que permitan lanzar *kernels* que exploten los diferentes niveles de paralelismo que ofrecen las *GPU*s, intentar ocultar el tiempo dedicado a transferencias de memoria solapando tiempo de cómputo y de transferencia, etc.

Finalmente, queremos comentar que durante el desarrollo del proyecto hemos aplicado una gran diversidad de conocimientos aprendidos a lo largo de la carrera. Destacamos la asignatura de *Compiladores*, fundamental para desarrollar cualquier compilador, ya sea a nivel *source-to-source* o binario. La asignatura de *Sistemas Operativos* nos ha aportado conocimientos sobre el control de los hilos de ejecución, fundamental para desarrollar nuestro *runtime*. Y para acabar, *Arquitectura de Computadores* nos ha permitido indagar y entender las diferencias en las arquitecturas de las *GPU*s y *CPU*s con sus beneficios y limitaciones.

APÉNDICES

Apéndice A

Resultados de la ejecución

CLInfo

CLInfo es un programa incluido dentro del *SDK* de *ATI* que nos permite ver las caracterísitcas de los dispositivos de *OpenCL* disponibles en el sistema.

A.1. CLInfo del ordenador portátil

A continuación presentamos el resultado de la ejecución del *CLInfo* para el entorno definido en la tabla 7.1.

```
1  Number of platforms:                                 2
2    Platform Profile:                                  FULL_PROFILE
3    Platform Version:                                  OpenCL 1.1 AMD-
       APP-SDK-v2.4 (595.10)
4    Platform Name:                                     AMD Accelerated
       Parallel Processing
5    Platform Vendor:                                   Advanced Micro
       Devices, Inc.
6    Platform Extensions:                               cl_khr_icd
       cl_amd_event_callback cl_amd_offline_devices
7    Platform Profile:                                  FULL_PROFILE
8    Platform Version:                                  OpenCL 1.0 CUDA
       3.2.1
9    Platform Name:                                     NVIDIA CUDA
10   Platform Vendor:                                   NVIDIA
       Corporation
11   Platform Extensions:
       cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing
       cl_nv_compiler_options cl_nv_device_attribute_query
       cl_nv_pragma_unroll
12
13
14   Platform Name:                                     AMD Accelerated
       Parallel Processing
15 Number of devices:                                   1
16   Device Type:
       CL_DEVICE_TYPE_CPU
17   Device ID:                                         4098
18   Max compute units:                                 4
19   Max work items dimensions:                         3
20     Max work items[0]:                               1024
21     Max work items[1]:                               1024
22     Max work items[2]:                               1024
23   Max work group size:                               1024
24   Preferred vector width char:                       16
25   Preferred vector width short:                      8
26   Preferred vector width int:                        4
27   Preferred vector width long:                       2
28   Preferred vector width float:                      4
29   Preferred vector width double:                     0
30   Max clock frequency:                               1197Mhz
31   Address bits:                                      32
32   Max memory allocation:                             1073741824
33   Image support:                                     Yes
```

```
34  Max number of images read arguments:                128
35  Max number of images write arguments:               8
36  Max image 2D width:                                 8192
37  Max image 2D height:                                8192
38  Max image 3D width:                                 2048
39  Max image 3D height:                                2048
40  Max image 3D depth:                                 2048
41  Max samplers within kernel:                         16
42  Max size of kernel argument:                        4096
43  Alignment (bits) of base address:                   1024
44  Minimum alignment (bytes) for any datatype:         128
45  Single precision floating point capability
46    Denorms:                                          Yes
47    Quiet NaNs:                                       Yes
48    Round to nearest even:                            Yes
49    Round to zero:                                    Yes
50    Round to +ve and infinity:                        Yes
51    IEEE754-2008 fused multiply-add:                  No
52  Cache type:                                         Read/Write
53  Cache line size:                                    64
54  Cache size:                                         32768
55  Global memory size:                                 3758096384
56  Constant buffer size:                               65536
57  Max number of constant args:                        8
58  Local memory type:                                  Global
59  Local memory size:                                  32768
60  Error correction support:                           0
61  Profiling timer resolution:                         1
62  Device endianess:                                   Little
63  Available:                                          Yes
64  Compiler available:                                 Yes
65  Execution capabilities:
66    Execute OpenCL kernels:                           Yes
67    Execute native function:                          Yes
68  Queue properties:
69    Out-of-Order:                                     No
70    Profiling :                                       Yes
71  Platform ID:                                        0xb7489f20
72  Name:                                               Intel(R) Core(TM)
        i5 CPU       M 540  @ 2.53GHz
73  Vendor:                                             GenuineIntel
74  Driver version:                                     2.0
75  Profile:                                            FULL_PROFILE
76  Version:                                            OpenCL 1.1 AMD-
      APP-SDK-v2.4 (595.10)
77  Extensions:                                         cl_khr_fp64
      cl_amd_fp64 cl_khr_global_int32_base_atomics
      cl_khr_global_int32_extended_atomics
      cl_khr_local_int32_base_atomics
      cl_khr_local_int32_extended_atomics
      cl_khr_byte_addressable_store cl_khr_gl_sharing
      cl_ext_device_fission cl_amd_device_attribute_query
      cl_amd_vec3 cl_amd_media_ops cl_amd_popcnt cl_amd_printf
```

```
78
79
80    Platform Name:                                    NVIDIA CUDA
81  Number of devices:                                  1
82    Device Type:
          CL_DEVICE_TYPE_GPU
83    Device ID:                                        4318
84    Max compute units:                                6
85    Max work items dimensions:                        3
86      Max work items[0]:                              512
87      Max work items[1]:                              512
88      Max work items[2]:                              64
89    Max work group size:                              512
90    Preferred vector width char:                      1
91    Preferred vector width short:                     1
92    Preferred vector width int:                       1
93    Preferred vector width long:                      1
94    Preferred vector width float:                     1
95    Preferred vector width double:                    0
96    Max clock frequency:                              1100Mhz
97    Address bits:                                     4724464025632
98    Max memory allocation:                            134217728
99    Image support:                                    Yes
100   Max number of images read arguments:              128
101   Max number of images write arguments:             8
102   Max image 2D width:                               4096
103   Max image 2D height:                              32768
104   Max image 3D width:                               2048
105   Max image 3D height:                              2048
106   Max image 3D depth:                               2048
107   Max samplers within kernel:                       16
108   Max size of kernel argument:                      4352
109   Alignment (bits) of base address:                 2048
110   Minimum alignment (bytes) for any datatype:       128
111   Single precision floating point capability
112     Denorms:                                        No
113     Quiet NaNs:                                     Yes
114     Round to nearest even:                          Yes
115     Round to zero:                                  Yes
116     Round to +ve and infinity:                      Yes
117     IEEE754-2008 fused multiply-add:                Yes
118   Cache type:                                       None
119   Cache line size:                                  0
120   Cache size:                                       0
121   Global memory size:                               536150016
122   Constant buffer size:                             65536
123   Max number of constant args:                      9
124   Local memory type:                                Scratchpad
125   Local memory size:                                16384
126   Error correction support:                         0
127   Profiling timer resolution:                       1000
128   Device endianess:                                 Little
129   Available:                                        Yes
```

```
130   Compiler available:                                   Yes
131   Execution capabilities:
132     Execute OpenCL kernels:                             Yes
133     Execute native function:                            No
134   Queue properties:
135     Out-of-Order:                                       Yes
136     Profiling :                                         Yes
137   Platform ID:                                          0x9ee42a0
138   Name:                                                 GeForce GT 330M
139   Vendor:                                               NVIDIA
        Corporation
140   Driver version:                                       260.19.26
141   Profile:                                              FULL_PROFILE
142   Version:                                              OpenCL 1.0 CUDA
143   Extensions:
        cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing
        cl_nv_compiler_options cl_nv_device_attribute_query
        cl_nv_pragma_unroll  cl_khr_global_int32_base_atomics
        cl_khr_global_int32_extended_atomics
        cl_khr_local_int32_base_atomics
        cl_khr_local_int32_extended_atomics
```

A.2. CLInfo del ordenador de escritorio

A continuación presentamos el resultado de la ejecución del *CLInfo* para el entorno definido en la tabla 7.2.

```
1  Number of platforms:                                 2
2    Platform Profile:                                  FULL_PROFILE
3    Platform Version:                                  OpenCL 1.0 CUDA
       4.0.1
4    Platform Name:                                     NVIDIA CUDA
5    Platform Vendor:                                   NVIDIA
       Corporation
6    Platform Extensions:
       cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing
       cl_nv_compiler_options cl_nv_device_attribute_query
       cl_nv_pragma_unroll
7    Platform Profile:                                  FULL_PROFILE
8    Platform Version:                                  OpenCL 1.1 AMD-
       APP-SDK-v2.4 (595.10)
9    Platform Name:                                     AMD Accelerated
       Parallel Processing
10   Platform Vendor:                                   Advanced Micro
       Devices, Inc.
11   Platform Extensions:                               cl_khr_icd
       cl_amd_event_callback cl_amd_offline_devices
12
13
14   Platform Name:                                     NVIDIA CUDA
15 Number of devices:                                   2
16   Device Type:
       CL_DEVICE_TYPE_GPU
17   Device ID:                                         4318
18   Max compute units:                                 14
19   Max work items dimensions:                         3
20     Max work items[0]:                               1024
21     Max work items[1]:                               1024
22     Max work items[2]:                               64
23   Max work group size:                               1024
24   Preferred vector width char:                       1
25   Preferred vector width short:                      1
26   Preferred vector width int:                        1
27   Preferred vector width long:                       1
28   Preferred vector width float:                      1
29   Preferred vector width double:                     1
30   Max clock frequency:                               1250Mhz
31   Address bits:                                      5368709120032
32   Max memory allocation:                             335429632
33   Image support:                                     Yes
34   Max number of images read arguments:               128
```

```
35  Max number of images write arguments:          8
36  Max image 2D width:                            4096
37  Max image 2D height:                           32768
38  Max image 3D width:                            2048
39  Max image 3D height:                           2048
40  Max image 3D depth:                            2048
41  Max samplers within kernel:                    16
42  Max size of kernel argument:                   4352
43  Alignment (bits) of base address:              4096
44  Minimum alignment (bytes) for any datatype:    128
45  Single precision floating point capability
46    Denorms:                                     Yes
47    Quiet NaNs:                                  Yes
48    Round to nearest even:                       Yes
49    Round to zero:                               Yes
50    Round to +ve and infinity:                   Yes
51    IEEE754-2008 fused multiply-add:             Yes
52  Cache type:                                    Read/Write
53  Cache line size:                               128
54  Cache size:                                    229376
55  Global memory size:                            1341718528
56  Constant buffer size:                          65536
57  Max number of constant args:                   9
58  Local memory type:                             Scratchpad
59  Local memory size:                             49152
60  Error correction support:                      0
61  Profiling timer resolution:                    1000
62  Device endianess:                              Little
63  Available:                                     Yes
64  Compiler available:                            Yes
65  Execution capabilities:
66    Execute OpenCL kernels:                      Yes
67    Execute native function:                     No
68  Queue properties:
69    Out-of-Order:                                Yes
70    Profiling :                                  Yes
71  Platform ID:                                   0x9c89490
72  Name:                                          GeForce GTX 470
73  Vendor:                                        NVIDIA
        Corporation
74  Driver version:                                270.41.06
75  Profile:                                       FULL_PROFILE
76  Version:                                       OpenCL 1.0 CUDA
77  Extensions:
        cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing
        cl_nv_compiler_options cl_nv_device_attribute_query
        cl_nv_pragma_unroll  cl_khr_global_int32_base_atomics
        cl_khr_global_int32_extended_atomics
        cl_khr_local_int32_base_atomics
        cl_khr_local_int32_extended_atomics cl_khr_fp64
78
79
```

```
80  Device Type:
        CL_DEVICE_TYPE_GPU
81  Device ID:                                  4318
82  Max compute units:                          14
83  Max work items dimensions:                  3
84    Max work items[0]:                        512
85    Max work items[1]:                        512
86    Max work items[2]:                        64
87  Max work group size:                        512
88  Preferred vector width char:                1
89  Preferred vector width short:               1
90  Preferred vector width int:                 1
91  Preferred vector width long:                1
92  Preferred vector width float:               1
93  Preferred vector width double:              0
94  Max clock frequency:                        1500Mhz
95  Address bits:                               6442450944032
96  Max memory allocation:                      134217728
97  Image support:                              Yes
98  Max number of images read arguments:        128
99  Max number of images write arguments:       8
100 Max image 2D width:                         4096
101 Max image 2D height:                        32768
102 Max image 3D width:                         2048
103 Max image 3D height:                        2048
104 Max image 3D depth:                         2048
105 Max samplers within kernel:                 16
106 Max size of kernel argument:                4352
107 Alignment (bits) of base address:           2048
108 Minimum alignment (bytes) for any datatype: 128
109 Single precision floating point capability
110   Denorms:                                  No
111   Quiet NaNs:                               Yes
112   Round to nearest even:                    Yes
113   Round to zero:                            Yes
114   Round to +ve and infinity:                Yes
115   IEEE754-2008 fused multiply-add:          Yes
116 Cache type:                                 None
117 Cache line size:                            0
118 Cache size:                                 0
119 Global memory size:                         536674304
120 Constant buffer size:                       65536
121 Max number of constant args:                9
122 Local memory type:                          Scratchpad
123 Local memory size:                          16384
124 Error correction support:                   0
125 Profiling timer resolution:                 1000
126 Device endianess:                           Little
127 Available:                                  Yes
128 Compiler available:                         Yes
129 Execution capabilities:
130   Execute OpenCL kernels:                   Yes
131   Execute native function:                  No
```

```
132   Queue properties:
133     Out-of-Order:                                 Yes
134     Profiling :                                   Yes
135   Platform ID:                                    0x9c89490
136   Name:                                           GeForce 9800 GT
137   Vendor:                                         NVIDIA
        Corporation
138   Driver version:                                 270.41.06
139   Profile:                                        FULL_PROFILE
140   Version:                                        OpenCL 1.0 CUDA
141   Extensions:
        cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing
        cl_nv_compiler_options cl_nv_device_attribute_query
        cl_nv_pragma_unroll  cl_khr_global_int32_base_atomics
        cl_khr_global_int32_extended_atomics
142
143
144   Platform Name:                                  AMD Accelerated
        Parallel Processing
145 Number of devices:                                1
146   Device Type:
        CL_DEVICE_TYPE_CPU
147   Device ID:                                      4098
148   Max compute units:                              4
149   Max work items dimensions:                      3
150     Max work items[0]:                            1024
151     Max work items[1]:                            1024
152     Max work items[2]:                            1024
153   Max work group size:                            1024
154   Preferred vector width char:                    16
155   Preferred vector width short:                   8
156   Preferred vector width int:                     4
157   Preferred vector width long:                    2
158   Preferred vector width float:                   4
159   Preferred vector width double:                  0
160   Max clock frequency:                            2003Mhz
161   Address bits:                                   32
162   Max memory allocation:                          1073741824
163   Image support:                                  Yes
164   Max number of images read arguments:            128
165   Max number of images write arguments:           8
166   Max image 2D width:                             8192
167   Max image 2D height:                            8192
168   Max image 3D width:                             2048
169   Max image 3D height:                            2048
170   Max image 3D depth:                             2048
171   Max samplers within kernel:                     16
172   Max size of kernel argument:                    4096
173   Alignment (bits) of base address:               1024
174   Minimum alignment (bytes) for any datatype:     128
175   Single precision floating point capability
176     Denorms:                                      Yes
177     Quiet NaNs:                                   Yes
```

```
178     Round to nearest even:                          Yes
179     Round to zero:                                  Yes
180     Round to +ve and infinity:                      Yes
181     IEEE754-2008 fused multiply-add:                No
182   Cache type:                                       Read/Write
183   Cache line size:                                  64
184   Cache size:                                       32768
185   Global memory size:                               3758096384
186   Constant buffer size:                             65536
187   Max number of constant args:                      8
188   Local memory type:                                Global
189   Local memory size:                                32768
190   Error correction support:                         0
191   Profiling timer resolution:                       1
192   Device endianess:                                 Little
193   Available:                                        Yes
194   Compiler available:                               Yes
195   Execution capabilities:
196     Execute OpenCL kernels:                         Yes
197     Execute native function:                        Yes
198   Queue properties:
199     Out-of-Order:                                   No
200     Profiling :                                     Yes
201   Platform ID:                                      0xb6b80f20
202   Name:                                             Intel(R) Core(TM)
        2 Quad CPU     Q9550  @ 2.83GHz
203   Vendor:                                           GenuineIntel
204   Driver version:                                   2.0
205   Profile:                                          FULL_PROFILE
206   Version:                                          OpenCL 1.1 AMD-
        APP-SDK-v2.4 (595.10)
207   Extensions:                                       cl_khr_fp64
        cl_amd_fp64 cl_khr_global_int32_base_atomics
        cl_khr_global_int32_extended_atomics
        cl_khr_local_int32_base_atomics
        cl_khr_local_int32_extended_atomics
        cl_khr_byte_addressable_store cl_khr_gl_sharing
        cl_ext_device_fission cl_amd_device_attribute_query
        cl_amd_vec3 cl_amd_media_ops cl_amd_popcnt cl_amd_printf
```

Bibliografía

[1] Barcelona supercomputing center performance tools. http://www.bsc.es/ssl/apps/performanceTools/.

[2] Nvidia cuda. http://www.nvidia.com/object/cuda_home.html.

[3] Online forum, amd: Using multiple gpus in a opencl program. http://forums.amd.com/devforum/messageview.cfm?catid=390&threadid=129284.

[4] Opencl fission device extension. http://www.khronos.org/registry/cl/extensions/ext/cl_ext_device_fission.txt.

[5] OpenMP Homepage. http://openmp.org/.

[6] *Official OpenMP Specification - Version 3.0*, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[7] Inc. Advanced Micro Devices. *AMD APP SDK v2.4 - OpenCL FAQ*, 2011. http://developer.amd.com/sdks/amdappsdk/assets/AMD_APP_SDK_FAQ.pdf.

[8] Rosa M. Badia. Easy programming heterogeneous systems: Starss presentation and complexhpc, May 2011. http://www.cs.vu.nl/complexhpc2011/slides/complexHPC2011-StarSs.pdf.

[9] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a Proposal for Programming Heterogeneous Multi-core Architectures.

[10] Rob Farber. Rob farber's massively parallel programming series. part 4: Coordinating computations with opencl queues. http://www.codeproject.com/KB/showcase/OpenCL-Queues.aspx?display=PrintAll.

[11] Khronos Group. Opencl specification. http://www.khronos.org/opencl.

[12] John C. Hull. *Options, Futures and Other Derivatives.* Prentice Hall, 5th edition, July 2002.

[13] Ken Perlin. Improving Noise. *SIGGRAPH*, 2002. http://mrl.nyu.edu/~perlin/paper445.pdf.

[14] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, March 2005. http://www.gotw.ca/publications/concurrency-ddj.htm.